# Design of Robust Scheduling Methodologies for High Performance Computing

**Inauguraldissertation**

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

Ali Omar Abdelazim Mohammed

aus Ägypten

Basel, 2020

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Florina M. Ciorba
Prof. Dr. Felix Wolf

Basel, den 17.12.2019

Prof. Dr. Martin Spiess, Dekan

# Abstract

Scientific applications are often large, complex, computationally-intensive, and irregular. Loops are often an abundant source of parallelism in scientific applications. Due to the ever-increasing computational needs of scientific applications, high performance computing (HPC) systems have become larger and more complex, offering increased parallelism at multiple hardware levels.

Load imbalance, caused by irregular computational load per task and unpredictable computing system characteristics (system variability), often degrades the performance of applications. Besides, perturbations, such as reduced computing power, network latency availability, or failures, can severely impact the performance of the applications. System variability and perturbations are only expected to increase in future extreme-scale computing systems. Extrapolating the current failure rate to Exascale would result in a failure every 20 minutes. Such failure rate and perturbations would render the computing systems unusable. This doctoral thesis improves the performance of computationally-intensive scientific applications on HPC systems via robust load balancing. Robust scheduling ensures and maintains improved load balanced execution under unpredictable application and system characteristics.

A number of dynamic loop self-scheduling (DLS) techniques have been introduced and successfully used in scientific applications between the 1980s and 2000s. These DLS techniques are not fault-tolerant as they were originally introduced. In this thesis, we identify three major research questions to achieve robust scheduling (1) How to ensure that the DLS techniques employed in scientific applications today adhere to their original design goals and specifications? (2) How to select a DLS technique that will achieve improved performance under perturbations? (3) How to tolerate perturbations during execution and maintain a load balanced execution on HPC systems?

To answer the first question, we reproduced the original experiments that introduced the DLS techniques to verify their present implementation. Simulation is used to reproduce experiments on systems from the past. *Realistic simulation induces a similar analysis and conclusions to the analysis of the native results.* To this end, we devised an approach for bridging the native and simulative executions of parallel applications on HPC systems. This simulation approach is used to reproduce scheduling experiments on past and present systems to verify the implementation of DLS techniques.

Given the multiple levels of parallelism offered by the present HPC systems, we analyzed the load imbalance in scientific applications, from computer vision, astrophysics, and mathematical kernels, at both thread and process levels. This analysis revealed a

significant interplay between thread level and process level load balancing. We found that dynamic load balancing at the thread level propagates to the process level and vice versa. However, the best application performance is only achieved by two-level dynamic load balancing.

Next, we examined the performance of applications under perturbations. We found that the most robust DLS technique does not deliver the best performance under various perturbations. The most efficient DLS technique changes by changing the application, the system, or perturbations during execution. This signifies the algorithm selection problem in the DLS. We leveraged realistic simulations to address the algorithm selection problem of scheduling under perturbations via a simulation assisted approach (SimAS), which answers the second question. SimAS dynamically selects DLS techniques that improve the performance depending on the application, system, and perturbations during the execution.

To answer the third question, we introduced a robust dynamic load balancing (rDLB) approach for the robust self-scheduling of scientific applications under failures (question 3). rDLB proactively reschedules already allocated tasks and requires no detection of perturbations. rDLB tolerates up to $P-1$ processor failures ($P$ is the number of processors allocated to the application) and boosts the flexibility of applications against nonfatal perturbations, such as reduced availability of resources.

This thesis is the first to provide insights into the interplay between thread and process level dynamic load balancing in scientific applications. Verified DLS techniques, SimAS, and rDLB are integrated into an MPI-based dynamic load balancing library (*DLS4LB*), which supports thirteen DLS techniques, for robust dynamic load balancing of scientific applications on HPC systems. Using the methods devised in this thesis, we improved the performance of scientific applications by up to 21% via two-level dynamic load balancing. Under perturbations, we enhanced their performance by a factor of 7 and their flexibility by a factor of 30. This thesis opens up the horizons into understanding the interplay of load balancing between various levels of software parallelism and lays the ground for robust multilevel scheduling for the upcoming Exascale HPC systems and beyond.

# Acknowledgments

I would like to express my appreciation to my research advisor Professor Florina M. Ciorba. She has helped me expand and develop my knowledge throughout the years of my doctorate studies and provided insightful, timely, and valuable guidance and resources. I very much appreciate her patience during our long, fruitful discussions and her constant support and encouragement throughout the work of this thesis.

I would like to thank Professor Heiko Schuldt sincerely for his warm support and kindness. His guidance and strong support were vital in completing my doctorate studies.

Special thanks to Professor Ioana Banicescu for her in-depth insights and feedback throughout our collaboration. Acknowledgments go to the support of my friends and colleagues. Many thanks to Antonio Maffia and Danilo Guerrera, who acted as my big brothers during the first months of my studies. Thanks to Ahmed Eleliemy, whom his presence made me feel at home. I would like to thank also Aurélien Cavelan for many stimulating discussions and generous advice. Thanks Jonas Müller Korndörfer for being such a good friend and for the joy and warmth you bring in the group. Acknowledgments also go to Michal Grabarczyk, Robert Frank, Bas Kin, and Rubén Cabezón for many fruitful discussions. Thanks to all my friends, your wonderful friendship made my studies enjoyable experience and helped me through the toughest days.

Most of all, I am very grateful for my parents, my brothers, and my sister. Completing my studies would not have been possible without your love, support, and encouragement. Special thanks to my wife, Nour, who shares with me my dreams and supports me with her love and understanding to achieve them. Finally, to the latest newcomer in my family, Laila, you are the joy in my life.

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

Scientific computing is the cornerstone of scientific research nowadays. Numerical simulations, computational data analysis, and modeling are becoming a crucial instrument in various scientific and engineering fields [TMT+19; LA18]. Scientists and engineers consider computers as "universal instruments of insight" [DGK19]. From predictive multiscale models of organisms behaviors in Biology [LA18] to type Ia supernovae explosion simulations in Astrophysics [PLF+07], scientific applications are growing in their complexity and computational needs.

Computers mainly consist of transistors that are combined to form processing elements (PEs). PEs and memory units (cache memory) are placed on chips that represent the basic building blocks of computers. In 1965, Gordon Moore anticipated that the number of transistors per chip is going to double every year due to the advancements in the transistor fabrication technology [Moo+65], which is known as Moore's law. In 1975, he revised his prediction to be doubling the numbers per chip every two years. Scaling down transistor feature size allowed the scaling up of its operating frequency, therefore, increasing the performance of applications for free. By the beginning of the 2000s, Moore's law would not hold longer due to the thermal and physical properties of transistor fabrication technologies [Sch97; Kis02]. Operating frequencies can not be scaled up, and, consequently, applications performance can no longer be improved for free. Therefore, parallel processing is the gateway for scientific applications to achieve high performance.

Modern high performance computing (HPC) systems provide a high level of hardware parallelism at multiple levels (instruction, core, socket, and node) to boost their offered computational power. HPC systems are constructed by connecting many computer chips to form nodes, racks, and supercomputing systems (see Figure 1.1). PEs on the same compute node share their main memory, i.e., *shared memory systems*. Generally, multiple nodes do not share their memory. Therefore, the system memory is

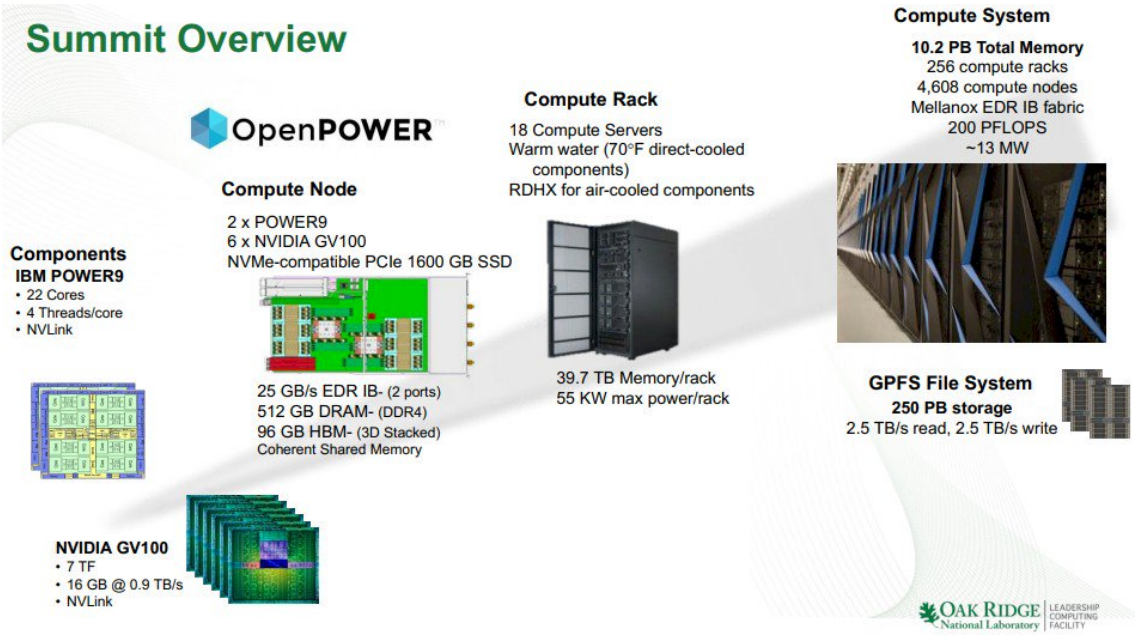



**Figure 1.1 Oak Ridge National Laboratory (ORNL) Summit Supercomputer Architecture.** Summit [Ser19] is the top supercomputer in the top500 supercomputers June 2019 list with a peak performance of 200 PetaFLOP/s.

distributed across nodes. Hence the term *distributed memory systems* or *distributed systems* is often used to refer to multiple nodes systems that may be in the same geographic location or not. Supercomputers are typically constructed by interconnecting multiple high-performance compute nodes, via interconnection networks, together with storage to form one large high performance distributed computing system. For example, the number of central processing unit (CPU) cores per node in the top 3 supercomputing systems[1] ranges from 44 to 260 cores and around 4000 nodes per system. Figures 1.2 to 1.4 show the evolution of the top HPC system in the top500 list over the years in terms of number of cores per node, number of nodes per system, and the overall number of cores per system. The large number of cores and nodes per system represents a significant challenge for scientific applications to harness such massive compute-power effectively.

For all such high parallelism and high performance of modern HPC systems, they do not fulfill the increasing computational needs of scientific applications, in certain cases. For example, state-of-the-art weather prediction models execute $100 - 250$ times slower than required to achieve predictions with 1 km resolution on current Petascale ($10^{15}$ floating-point operations per second, FLOP/s) machines [SBW+19]. Therefore, HPC systems keep growing more massive and more complex to fulfill the ever-increasing needs of scientific applications.

[1] https://www.top500.org/list/2019/06/

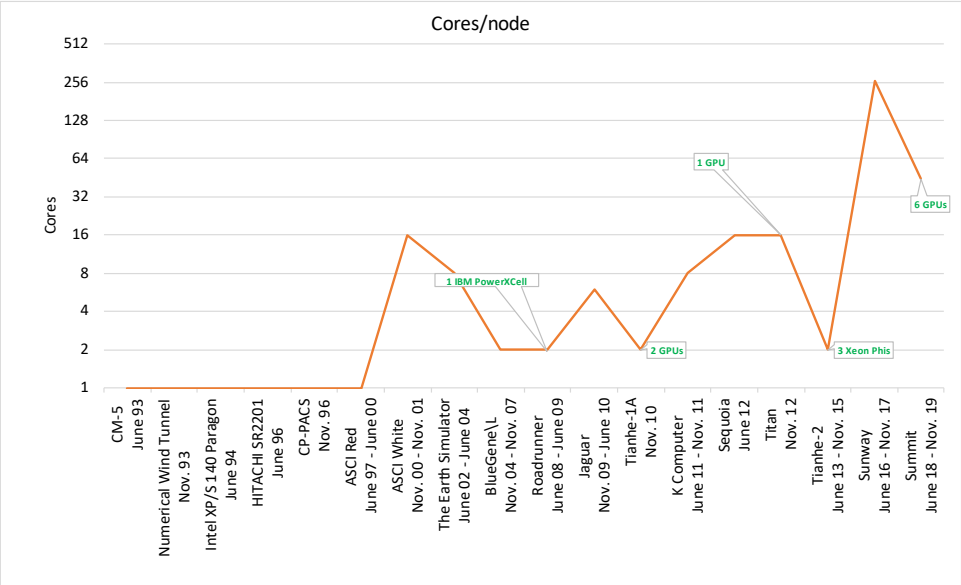



**Figure 1.2 Number of processing cores per node in the top HPC systems according to the top500 list over the years.** Accelerators in top systems are shown as green notes. The number of cores per node is increasing, specifically in the few previous years. Modern HPC systems contain tens to hundreds of cores per compute node.

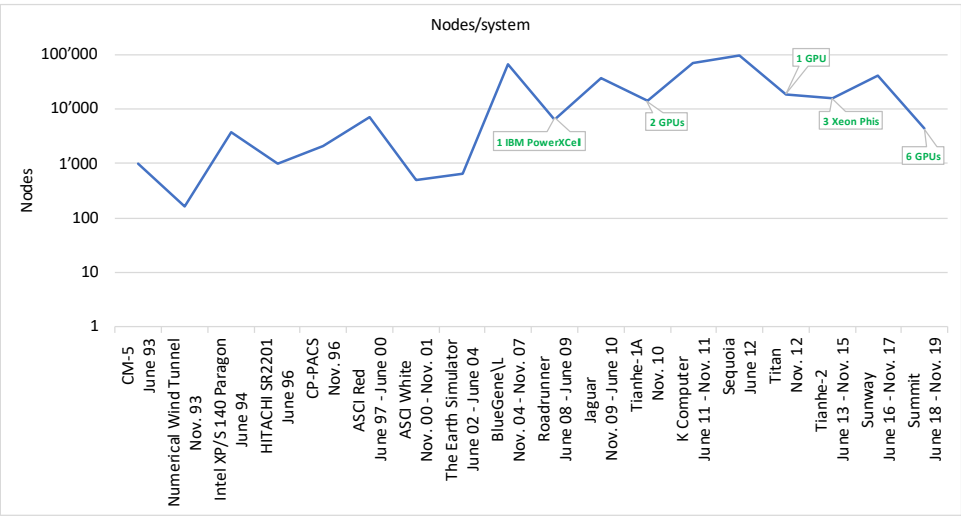



**Figure 1.3 Number of nodes per system in the top HPC systems according to the top500 list over the years.** Accelerators in top systems are shown as green notes. The number of nodes/system is significantly increasing, and it reaches tens of thousands of nodes in the modern HPC systems.

The current system sizes need to be scaled by a factor of 1000 to achieve Exascale performance. An Exascale HPC system refers to a system capable of achieving at least 1 ExaFLOP/s performance for the high performance Linpack [DLP03] (HPL) benchmark, that is one floating-point operation every billionth of a billionth of a second [SAB+19]. Europe [Kal19], the United States of America [KLQ19], China [QL18], and Japan [Sor19] each has planned to commission two or three Exascale ($10^{18}$ FLOP/s)

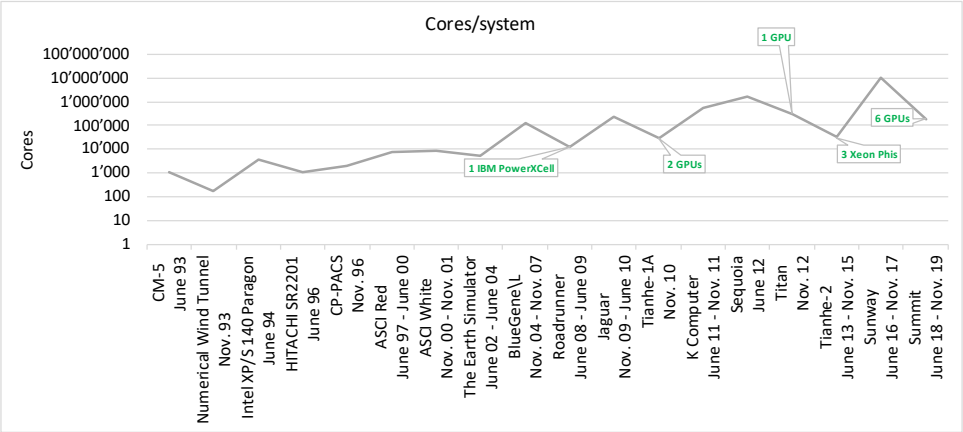



**Figure 1.4 Number of processing cores per system in the top HPC systems according to the top500 list over the years.** Accelerators in top systems are shown as green notes. The overall number of cores in a system is in the range of tens of millions of cores. This represents a significant challenge for scientific application to harness all the compute-power of this large number of cores effectively.

systems between 2020 and 2023. Such a large scale of HPC systems presents a challenge for scientific applications to harness the massive computing power of these systems.

## 1.1 Scientific Applications and their Performance Challenges

Applications need to exploit such high hardware parallelism of HPC systems by exposing and expressing software parallelism within themselves. Harnessing high parallel hardware computing power represents a challenge for applications to divide their workload into small chunks of work, i.e., tasks that are executed in parallel on multiple PEs. Large computationally-intensive parallel loops, in general, constitute the majority of the workload in scientific applications [ABC+06]. These loops are an abundant source of parallelism, as they typically contain a large number of computationally-intensive loop iterations, which are, in general, independent (or can be rendered independent via loop transformations) computational tasks that can be performed in parallel on multiple PEs.

Another challenge is the *scheduling* of application tasks, which is the distribution of application tasks and their mapping in space (to PEs) and in time (when to start a task on a PE) to work in parallel. Scheduling needs to assign tasks to PEs in such a way that balances the load among PEs with minimum overhead to minimize the overall application execution time.

Load imbalance between various PEs executing an application degrades its per-

formance and scalability. Scheduling and load balancing are among the most critical challenges identified on the road to Exascale systems [BBC+]. Load imbalance manifests due to application-related factors, such as nonuniform workload per task due to branches and conditional statements, or system-related factors, such as irregular PE performance, nonuniform memory and cache access times, and perturbations [TSL+17; BC05; VLWB+19].

Due to the hybrid nature of current HPC systems, distributed memory across compute nodes and shared memory within single nodes, hybrid parallelization of scientific applications at process level and thread level using MPI+OpenMP is the most common and successful approach [JJM+11; RHJ09; SB01]. Therefore, load balancing methods are essential at both process and thread levels.

Perturbations during application execution degrade its performance and lead to load imbalance as well. Nonfatal perturbations (see Section 3.1) during execution, due to system interference, error recovery, and error reporting, significantly affect application performance [VLWB+19; GSG+16; AE18]. Additionally, PE, node, and network failures (fatal perturbations) are expected to increase proportional to the number of components in a system [BBC+; WLB+16], i.e., CPU socket count and memory modules. Extrapolating the current failure rates to Exascale systems would result in a failure every 24 minutes, which would be prohibitive to scientific applications [BBC+]. Therefore, applications need to acquire robust properties to address such unpredictable perturbations during execution. A robust application maintains its correct execution and (load balanced) performance under perturbations.

Inspecting the load balancing and robustness of OpenMP and MPI explicates the gap in this area (see Figures 1.5 and 1.6). In the OpenMP standard, only three scheduling methods are available, which may not be sufficient to provide improved load balanced execution of applications on modern and future manycore computing systems. Moreover, no fault-tolerance methods are offered by the OpenMP standard or the libraries that implement it.

For MPI, there are non standard MPI implementations that provide certain fault tolerance functionality, such as rMPI [FRO+11], redMPI [FME+12a], FT-MPI [FD00], and user-level fault mitigation (ULFM) [BBH+12]. However, no standard load balancing method is offered in MPI or any of its implementations.

In this thesis, we propose robust and dynamic load balancing methods and implement them in OpenMP and MPI. Also, we study and analyze the dynamic load balancing at the two levels, process and thread levels, implemented using MPI+OpenMP. Therefore, the goal of this doctoral thesis is *improving the performance of computationally-intensive scientific applications on HPC systems via robust load balanc-*

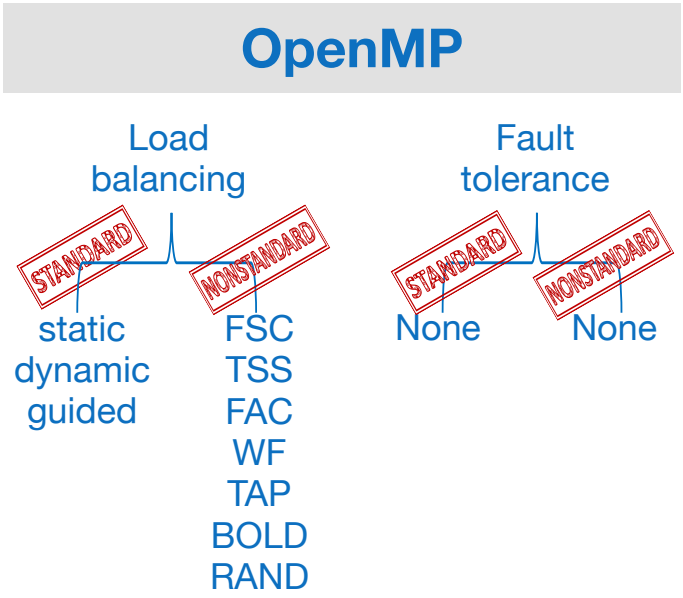



**Figure 1.5** **Load balancing and robustness in OpenMP.** It offers three scheduling methods for load balancing as a standard, which may be insufficient due to the increase in PEs count on modern shared-memory systems. Certain OpenMP runtime library implementations are extended with more DLS techniques (explained in detail in Chapters 2 and 9). No robustness methods are offered in OpenMP.

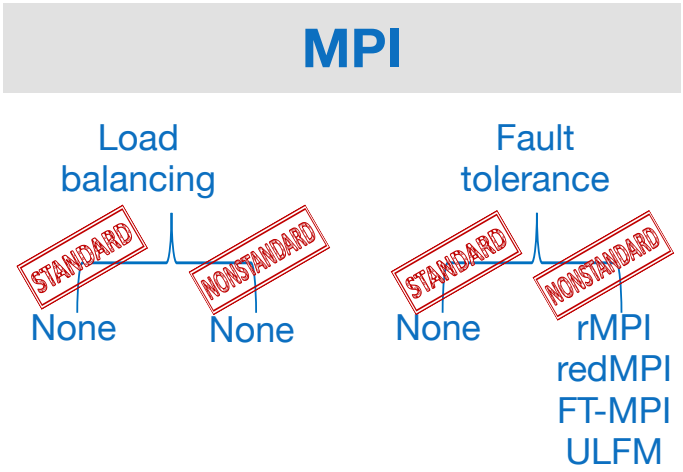



**Figure 1.6** **Load balancing and robustness in MPI.** No load balancing solutions is offered in the MPI standard or nonstandard implementations. For fault tolerance, there are several nonstandard MPI implementations that offer certain fault tolerance functionality, such as rMPI [FRO+11], redMPI [FME+12a], FT-MPI [FD00], and user-level fault mitigation (ULFM) [BBH+12]

*ing that maintains improved balanced load execution under unpredictable application and system characteristics.*

## 1.2 Research Problem and Questions

A number of dynamic loop self-scheduling (DLS) techniques (see Section 2.2) have been introduced and successfully used in scientific applications between the 1980s and 2000s (see Figure 1.7, DLS techniques explained in detail below in Section 2.2). These DLS techniques offer a broad spectrum of load balancing capabilities, including specific techniques that adapt to account for system-induced load imbalance. Such properties of DLS techniques makes them very suitable to achieve load balance on modern HPC systems.

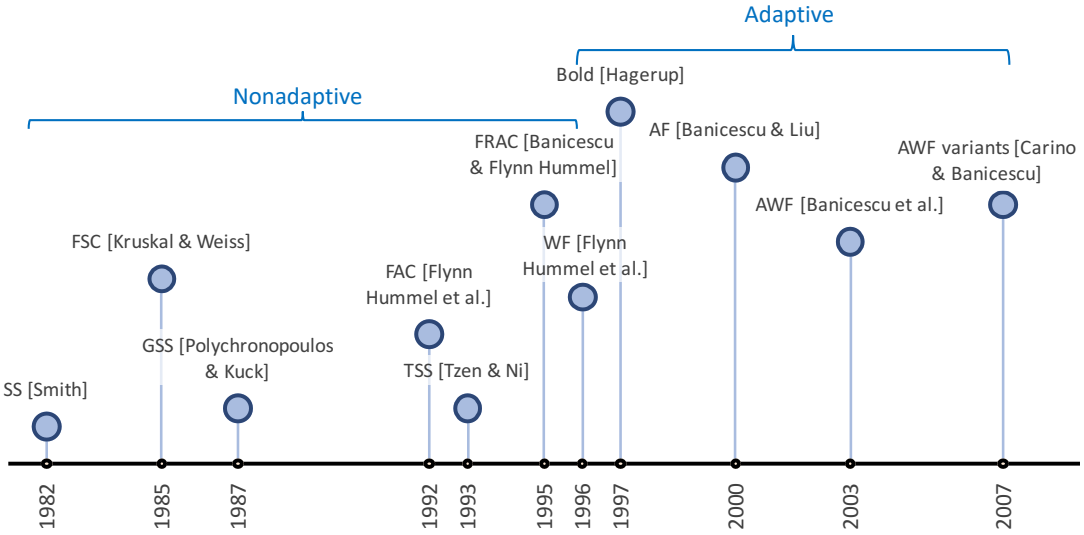



**Figure 1.7 A timeline of the most successful and adopted DLS techniques.** DLS techniques were introduced as mathematical formulae or pseudocodes. These techniques are not fault-tolerant as originally introduced.

Most of DLS techniques exist only as concepts as they were introduced, i.e., theoretically in publications and not as a dynamic load balancing library or part of a runtime. Also, the systems on which these techniques were developed and introduced no longer exist in most cases. This raises the first question (above in the Abstract) of

Q1 *How to ensure that the DLS techniques employed in scientific applications today adhere to their original design goals and specifications?*

Answering this question eliminates uncertainties regarding DLS implementation and their performance and establishes trustworthiness in their implementation.

The second question is

Q2 *How to select a DLS technique that will achieve improved performance under perturbations?*

Given the wide range of DLS techniques and the diversity of their characteristics and performance with different applications and systems, the selection of the most efficient DLS technique for an application-system pair is not trivial. Answering this question solves the algorithm selection problem for DLS.

DLS techniques, however efficient and adaptive, are not resilient by definition as they were introduced. They can not tolerate PE or network failures and results in degraded performance on highly perturbed systems. Failures (fatal perturbations) are only expected to increase in future HPC systems [BBC+; WLB+16]. Therefore, the third question is

Q3 *How to tolerate perturbations during execution and maintain a load balanced execution on HPC systems?*

Improving applications performance via load balanced execution under (nonfatal and fatal) perturbations achieves robust scheduling.

## 1.3 Approach

We describe our approach in addressing and answering the three identified research questions mentioned above. Figure 1.8 shows an overview of the employed approach and contributions of this doctoral thesis. The present implementation of DLS techniques needs to be verified against the original publications to answer the first research question. Reproduction, understood as revisiting a particular scientific problem without the original artifacts, contributes to the validation of those experiments, and establishes their scientific relevance [ACM16; HT13]. Given that the computing systems on which the DLS techniques were introduced and developed no longer exist, simulation is used for reproduction. Simulation is a crucial instrument of scientific research and helps mitigate experimentation cost and time. Moreover, performance simulation helps predict performance on existing and nonexisting (past or future) computing systems and permits experimentation in a controlled environment. Therefore, simulation is used for the reproduction experiments that verify the present DLS implementation.

Answering the first research question using reproduction via simulation requires realistic performance simulation of scheduling experiments. To this end, we propose methods for capturing applications and systems characteristics in simulation for the realistic performance simulation. We compare native and simulative performance results to evaluate how realistic are performance simulations and establish trust in simulation results. Realistic performance simulations are employed for the reproduction of

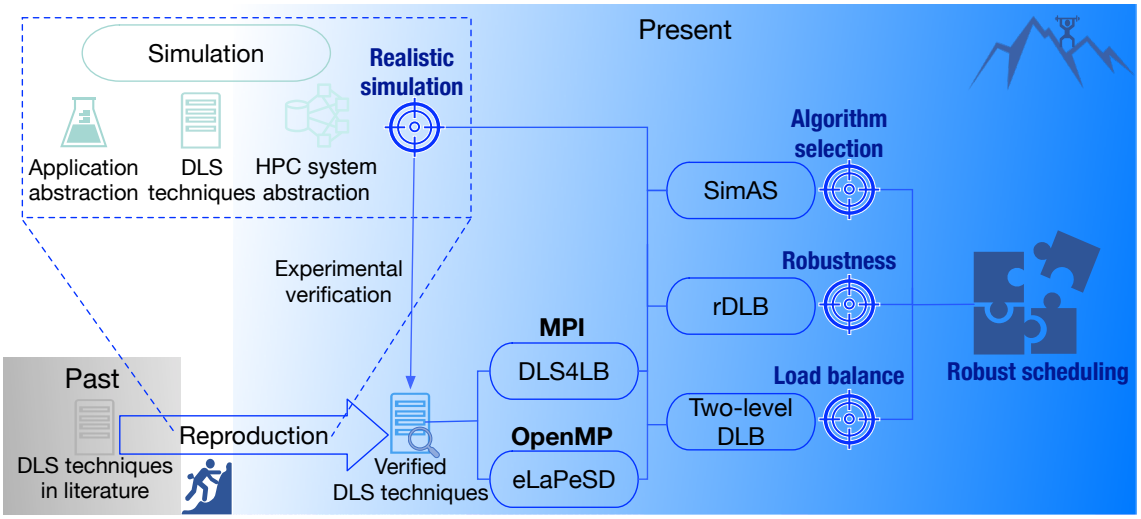



**Figure 1.8 Overview of the approach and contributions of this thesis.** Reproduction is employed for the verification of DLS techniques implementation. Realistic simulation methods are devised for performance simulation with DLS. Verified DLS techniques are implemented in OpenMP and MPI to enable applications with single- and two-level DLB. SimAS leverages realistic performance simulations for the dynamic selection of DLS techniques during execution. rDLB addresses robust scheduling under perturbations via proactive task re-execution. Realistic simulation, SimAS, rDLB, and two-level DLB are all essential for the robust scheduling on modern and future HPC systems.

scheduling experiments in original DLS publications for the verification of present DLS techniques implementation.

After the verification of the DLS techniques implementation, we investigate how to employ these techniques to scientific applications and how effective they are in improving application performance via load balancing. We show and discuss implementations of DLS techniques in OpenMP and MPI. DLS implementations in OpenMP and MPI enable scientific applications to employ dynamic load balancing (DLB) both at the thread level and process level. For the first time, we investigated the performance of scientific applications with DLB at the thread level only, process level only, and at both the thread and process levels simultaneously. We show the interplay between process level and thread level DLB and that load balancing effects propagate from one level to another.

We explore the performance of applications with various DLS techniques under perturbations to understand the impact of various perturbations on performance and answer the second research question. We confirm the hypothesis that *no single DLS technique achieves the best performance under different perturbations.* DLS techniques are designed to be very efficient in load balancing applications on HPC systems. However, they are not robust to different execution scenarios. To answer the second question, we devised a Simulation-assisted scheduling Algorithm Selection [MC18b] (SimAS) approach

for the dynamic selection of the most efficient DLS technique based on the system state and perturbations in the system. SimAS leverages realistic performance simulations for the selection of the most efficient DLS technique during execution.

We devised the robust Dynamic Load Balancing [MCC19] (rDLB) approach for the robust self-scheduling of applications on HPC systems to answer the third research question. rDLB tolerates $P-1$ failures, where $P$ is the number of PEs used by an application, and boosts applications performance and DLS techniques robustness in the presence of nonfatal perturbations.

The dynamic selection of DLS techniques during execution via SimAS, the robust scheduling with rDLB, two-level DLB, and realistic performance simulations are all necessary components for the robust scheduling and load balancing of scientific applications on modern and future HPC systems.

## 1.4 Contributions and Significance

The contributions of this doctoral thesis are summarized in the following:

**1. A realistic performance simulation approach for scientific applications with DLS.** We *detail* how to *capture application and computing system characteristics in simulation* for the realistic performance simulation [MEC+18a; MEC+19]. We *discuss and compare the influence* of different representations of application characteristics in simulation on the predicted simulative performance. We clarify how to *capture, fine-tune, and verify computing system representation* in simulation. Native and simulative applications performance are compared to evaluate *how realistic* is performance simulations, both, *quantitatively and qualitatively*. We *define a realistic performance simulation* as the simulation that leads to a *performance analysis* similar to the native one, and that *captures performance features* extracted from native performance results.

**2. A verification via reproduction method.** We devise a method for the *reproduction* of DLS experiments included in the literature *for the verification* of the present implementation of DLS techniques [MEC18]. We employ this method for the reproduction and verification of the factoring [FHSF92] DLS technique (FAC), which is one of the most successful DLS techniques. Verification of FAC lays the ground for the verification of *more complex DLS techniques*, such as adaptive weighted factoring [BVD03] (AWF), its variants [CB08], and adaptive factoring [BL00] (AF), which derive from FAC.

**3. Implementation of the DLS techniques.** We explore *centralized and decentralized* coordination approaches in implementing DLS techniques. We discuss the advantages and limitations of each approach. We present *DLS implementations* in the most successful and widely used programming models in HPC, namely open multiprocess-

ing (*OpenMP*) and message passing interface (*MPI*). This *encourages scientific applications developers* to employ DLS techniques, which is crucial for improving the performance and scalability of scientific applications on large-scale HPC systems.

**4. Single- and two-level dynamic load balancing of scientific applications.** We show how to use DLS techniques for the dynamic load balancing at a *single or two levels* of software parallelism. We *analyze load imbalance* in applications at *single and two levels*. We provide *insights into the interplay between the load balancing at two levels* and confirm that two-level dynamic load balancing ensures improved application performance.

**5. Simulation-assisted scheduling algorithm selection.** We analyze the *performance* of applications with different DLS techniques *under nonfatal perturbations* [MC18a]. We show that *DLS techniques* are very *efficient* in different execution scenarios, however, they are *not* very *robust*. We confirm the hypothesis that *no single technique* provides the *best performance* under all perturbations. We devise Simulation-assisted scheduling Algorithm Selection (SimAS) for the *dynamic selection* of DLS techniques under perturbations [MC18b]. SimAS *dynamically selects* DLS techniques where they *are the most efficient* to improve the performance of applications.

**6. Robust dynamic load balancing for parallel tasks.** To address *fatal and nonfatal* perturbations in large scale HPC systems, we devise robust Dynamic Load Balancing [MCC19] (rDLB). rDLB *proactively* reschedules tasks to tolerate fatal and nonfatal perturbations. It does *not require any fault detection* mechanism. Experimental evaluation of rDLB shows that it *tolerates* up to $P-1$ *failures*, where $P$ is the number of PEs allocated to the application and *boosts the robustness* of DLS techniques by up to a factor of 30 under nonfatal perturbations.

### 1.4.1 Significance

Realistic performance simulations are essential for predicting the performance of applications on present HPC systems. Thus, realistic simulations mitigate large (often infeasible) exploratory experiments to optimize application performance on a particular architecture. Also, realistic simulations predict performance on future nonexistent HPC systems, which helps to direct their design and optimizations.

Verified DLS techniques and their implementations in OpenMP and MPI encourage scientific application developers to employ DLS techniques in their codes. Thereby improving applications performance via single- and two-levels dynamic load balancing, as shown in this thesis. Increased PE count per node and increased number of nodes in large scale modern and future HPC systems, will only worsen the impact of load imbalance on applications performance. Verified DLS techniques at the thread level and

process level with *eLaPeSD* and *DLS4LB* address such problems.

SimAS addresses the problem of algorithm selection for DLS of scientific applications. In particular, it selects dynamically during execution the most efficient DLS techniques according to the application, system, and perturbations during execution. The dynamic selection of DLS techniques ensures the efficiency of DLS and improves applications performance under unexpected execution scenarios.

rDLB addresses both fatal and nonfatal perturbations and ensure the completion and the balanced load execution of scientific applications under perturbations.

Modern and future large scale HPC systems are expected to incur highly variable system performance and high failure rates (a failure every 24 minutes [BBC+]). Realistic simulations, verified DLS techniques implementations, SimAS, and rDLB all contribute to the robust scheduling of scientific applications and improve their performance on such systems. Without robust scheduling, modern and future HPC systems would be rendered unusable due to their high variability and failure rates.

## 1.5 Organization of the Thesis

The remainder of the thesis is structured into five parts. Part I contains an overview of the scientific background on loop scheduling and dynamic load balancing, faults, errors, failures, and performance simulation. In particular, a background on load imbalance, loop scheduling in general, and more specifics on DLS techniques are provided in Chapter 2. In Chapter 3, terms related to failures and robustness are defined. Perturbations in large scale HPC systems and their effect on performance are presented to identify the most critical perturbations to be considered in the subsequent chapters, which challenge applications performance. The SimGrid [CGL+14] simulation toolkit is introduced in Chapter 4, which is used for performance simulation of scientific applications. Part I ends with Chapter 5, which includes related work from literature on load balancing, fault tolerance, and simulation.

Part II contains our investigations and efforts in achieving realistic performance simulation and the verification of DLS techniques implementation via reproduction. Methods for the realistic performance simulation are presented in Chapter 6. In Chapter 7, the realistic simulation methods are used for the reproduction of DLS experiments from literature for the verification of the present implementation of DLS techniques. Various simulation approaches are compared to each other and against native performance in Chapter 8 to evaluate how realistic is performance simulations. Several application representations are compared for their impact on the predicted simulative performance.

Part III includes the implementation of DLS techniques in the most used programming models in HPC and the use of DLS techniques for single- and two-level dynamic load balancing. Centralized and decentralized implementations of DLS techniques and their implementations in OpenMP and MPI are discussed in Chapter 9. In Chapter 10, the load balancing of scientific applications at a single level using DLS techniques is presented. Also, the weak and strong scalability of scientific applications on homogeneous and heterogeneous HPC systems are discussed. The two-level load imbalance in scientific applications is analyzed in Chapter 11. Insights into the interplay between the dynamic load balancing at the two levels are provided based on the performance analysis.

Part IV includes our investigations and contributions on the robust scheduling and scheduling under perturbations. The performance of scientific applications with different DLS techniques under nonfatal perturbations is analyzed in Chapter 12. The SimAS method is presented for the dynamic selection of the most robust DLS technique. The rDLB approach is introduced for the robust scheduling against fatal and nonfatal perturbations in Chapter 13 .

Part V contains one chapter, Chapter 14, which concludes this doctoral thesis and highlights future outlook and open problems for future research.

## 1.6 Publications

Throughout the in-depth investigations, analysis, and development of robust scheduling in this doctoral thesis, many of our contributions received recognition from several prominent international conferences and journals in high performance computing. Below is the list of these publications:

[MCC+20] Mohammed A., Cavelan A., Ciorba F. M., Cabezón R. M., Banicescu I., "Two-level Dynamic Load Balancing for High Performance Scientific Applications". In the Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing 2020 (SIAM PP20), Feb. 2020, Seattle, WA, USA.

[MC19b] Mohammed A., Ciorba F. M., "SimAS: A Simulation-Assisted Approach for the Algorithm Selection Problem of Scheduling under Perturbations". In the Concurrency and Computation: Practice and Experience Journal, Special Issue on the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2018).

[MEC+19] Mohammed A., Eleliemy A., Ciorba F. M., Kasielke F., Banicescu I., "A Methodology for Realistically Simulating the Performance of Scientific Applications on

High Performance Computing Systems". In Future Generation Computer Systems Journal, On The Road to Exascale II Special Issue: Advances in High Performance Computing and Simulations.

[MCC19] Mohammed A., Cavelan A., Ciorba F. M., "rDLB: A Novel Approach for Robust Dynamic Load Balancing of Scientific Applications with Independent Tasks". In the Proceedings of the International Conference on High Performance Computing & Simulation (HPCS) 2019 July, Dublin, Ireland.

[MC18b] Mohammed A., Ciorba F. M., "SiL: An Approach for Adjusting Applications to Heterogeneous Systems Under Perturbations". In the Proceedings of European Conference on Parallel Processing Workshops (HeteroPar) 2018 Aug. 27 (pp. 456-468), Turin, Italy.

[MEC18] Mohammed A., Eleliemy A., Ciorba F. M., "Performance Reproduction and Prediction of Selected Dynamic Loop Scheduling Experiments". In the Proceedings of the International Conference on High Performance Computing & Simulation (HPCS) 2018 Jul. 16 (pp. 398-405), Orleans, France.

[MEC+18a] Mohammed A., Eleliemy A., Ciorba F. M., Kasielke F., Banicescu I., "Experimental Verification and Analysis of Dynamic Loop Scheduling in Scientific Applications". In the Proceedings of the $17^{th}$ International Symposium on Parallel and Distributed Computing (ISPDC) 2018 Jun. 25, Geneva, Switzerland.

In addition, other publications that were performed in collaboration with colleagues and are not directly related to this doctoral thesis are:

[GCC+19] Guerrera D., Cavelan A., Cabezón R. M., Imbert D., Piccinali J. G., Mohammed A., Mayer L., Reed D., and Ciorba F. M., "SPH-EXA: Enhancing the Scalability of SPH codes Via an Exascale-Ready SPH Mini-App". In the Proceedings of the Spheric International Workshop, 2019 June, Exeter, UK.

[EMC17a] Eleliemy A., Mohammed A., Ciorba F. M., "Efficient Generation of Parallel Spin-images Using Dynamic Loop Scheduling". In the Proceedings of the $19^{th}$ IEEE International Conference for High Performance Computing and Communications Workshops (HPCCW) 2017 Dec. 18, Bangkok, Thailand.

[EMC17b] Eleliemy A., Mohammed A., Ciorba F. M., "Exploring the Relation Between Two Levels of Scheduling Using a Novel Simulation Approach". In the Proceedings of the $16^{th}$ International Symposium on Parallel and Distributed Computing, 2017 Jul., Innsbruck, Austria.

Part I

# Foundations

# 2

# Load Imbalance and Loop Scheduling

Simulation and data are considered the third and fourth pillars of science after theory and experimentation. Scientific applications include weather prediction, N-body simulations, heat diffusion, and Monte-Carlo simulations, among others. They advance scientific and industrial fields, such as meteorology, chemistry, medicine, aviation, and astrophysics. These applications represent the typical workload of HPC systems. HPC systems have grown in the computational power and degrees of parallelism they offer to accommodate the ever-increasing computational demands of scientific applications.

In general, applications can be classified into computationally-intensive, communication-intensive, Input/output-intensive, or a combination of two or more of these. Computationally-intensive applications spend most of their execution time in computations (CPU time) whereas communication-intensive applications spend most of their execution time in communication either between compute nodes, CPU and memory (memory-bound), or CPU and input/output (including storage) devices (I/O bound). However, studying scientific applications revealed that there are seven numerical methods, known as Berkeley Dwarfs, that constitute most of the scientific applications [ABC+06]. These dwarfs are computationally-intensive in nature.

Computationally-intensive loops represent the main source of parallelism in scientific applications. Loops exist in scientific applications to iterate over the "physical simulated" domain and compute its properties. Loops are classified into *doall* and *doacross* loops [HLK+97]. *Doall* loops are loops where there is no dependency across loop iterations, i.e., its loop iterations are *embarrassingly parallel independent tasks. Doacross* loops are loops where there are dependencies between the loop iterations. *Doacross* loops are more complex to parallelize. However, synchronizations can be added to fulfill the dependencies across the loop iterations and partially parallelize independent loop iterations [CRA+08; PK87].

## 2.1 Load Imbalance

Load imbalance manifests as uneven finishing times of processing elements (PEs). It degrades the performance of scientific applications on HPC systems as faster PEs idly wait until slower PEs finish, instead of helping them finish faster [LHG+08]. Load imbalance is a major performance challenge for scientific applications, adversely affects scalability, and becomes more significant as the number of PEs increases, which is expected in the future Exascale systems and beyond [BWG12]. Load imbalance is caused by applications- and/or systemic-related characteristics [SMW18]. Application-induced load imbalance includes irregular computations per task due to conditional statements and branches. System-induced (i.e., system variability ) load imbalance includes nonuniform memory access (NUMA), nonuniform processing speed due to cache misses, architectural properties, or perturbations. For example, it was found that low-level optimization inside the CPU when one of the operands is zero causes load imbalance in a geophysics simulation application [TSL+17].

Load imbalance may manifest in more than one level of software parallelism, i.e., among processes (process-level) and threads (thread-level). For example, Figure 2.1 conceptually shows the two-level load imbalance of a scientific application parallelized using multiple processes and threads. Due to the thread level load imbalance, threads that finish early must wait until the slowest thread finishes (yellow regions). Therefore, the slowest thread dominates the performance of a process. Similarly, at the process level, the faster process has to wait for the slower ones, and the slowest process dominates the application performance. Load imbalance produces wait-states that can propagate across the application in the same level of parallelism (e.g., among threads only or processes only) and across different levels of parallelism (i.e., a wait state at one thread delays the execution of another process) [BGW+10; BGA+16]. The two-level load imbalance is a compound problem and not trivial to address as the scheduling performance at one level is influenced by the scheduling decisions at the other level. For example, the relation between batch-level and application-level scheduling was studied [EMC17b], and it was shown that a holistic solution that simultaneously addresses load imbalance at both levels results in better performance improvement than focusing on improving the performance at each level alone.

### 2.1.1 Load Balance Metrics

The load balance of the applications is measured by two metrics, namely the *coefficient of variation* (*c.o.v.*) of the parallel PEs finishing times [FHSF92] and the ratio of the mean PEs finishing times to the maximum PE finishing time (*mean/max*) [CBL08]. The c.o.v.

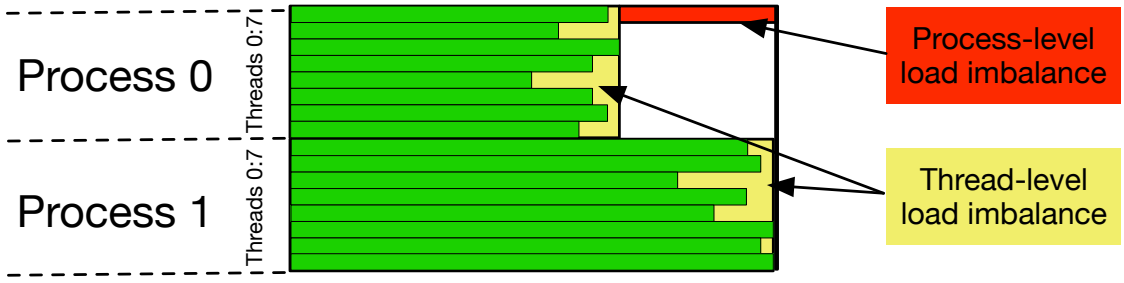



**Figure 2.1 Conceptual illustration of the impact of the two-level load imbalance of a scientific application on two processes, each with eight threads.** Due to the uneven execution progress at the thread level, faster threads wait for the slowest thread in each process, represented by the yellow regions. At the process level, process 0 (faster) waits for process 1 (slower), represented by the red region. The application completes when the slower process (process 1) finishes.

is calculated as the ratio between the standard deviation of PEs' finishing times to their mean value. Both metrics are unitless. A severe execution load imbalance corresponds to a c.o.v. $>> 0$ and a mean/max $<< 1$ while a negligible load imbalance corresponds to c.o.v. $\approx 0$ and mean/max $\approx 1$. The mean/max indicates how long the processes of an application have to wait for the slowest process due to load imbalance. A mean/max value of 1 represents a balanced load execution (lower bound), and a low value indicates that execution time is prolonged due to a process that lags behind all the other processes. When all processes except one have similar finishing times, the c.o.v. would have a low value and hides the fact that one process is lagging the execution, while it will show as a high value in mean/max metric (see Section 8.3.3).

## 2.2 Loop Scheduling

Scheduling, in the context of applications, is the mapping of applications tasks in space and time to PEs to minimize the application execution time. Scheduling is concerned with the assignment of tasks. The application itself generally distributes its data. Data can be centralized, distributed, or replicated. Certain scheduling techniques are locality-aware, i.e., try to assign tasks to PEs that already have the data for these tasks in their cache or local memory, to avoid data migration and improve performance.

In loop scheduling, each loop iteration is considered a task. Doall loops consist of parallel independent tasks. Scheduling methods aim to maximize the *load balance* between PEs with minimum *scheduling overhead* to minimize application execution time. *Scheduling overhead* includes the time taken to assign tasks to PEs, the time to calculate how many tasks to assign to a PE (chunk size), and synchronization and communication

overheads required to assign the tasks to PEs.

### 2.2.1 Work-sharing, Self-scheduling, and Work-stealing

Scheduling involves work partitioning and work assignment, i.e., how to divide the application workload into chunks and how to assign these work chunks to PEs. Scheduling is classified into three main categories based on the method of work assignment, namely

1. Work-sharing
2. Self-scheduling
3. Work-stealing.

In *work-sharing* all tasks are distributed among PEs as soon as they are created in such a way that achieves a balanced load among the PEs. In *self-scheduling* [BC05], whenever a PE is free, it allocates (or requests) a chunk of work (tasks) to execute from a *central work queue*. In *Work-stealing* [BL99], a work queue is associated with each PE. PEs execute tasks from their queue until it is empty. When a PE's queue is empty, it selects a *victim* PE and *steals* tasks from the victim's work queue to achieve load balance. *Self-scheduling* differs from work-stealing in that all PEs assign themselves (steal) work from a *central work queue*. Table 2.1 summarizes the differences between the three scheduling methods. This work studies and extends self-scheduling methods for robust application scheduling as they are proactive and can achieve improved load-balanced performance under perturbations.

From the task assignment point of view (when task assignment decisions are made), loop scheduling can be divided into static and dynamic techniques. Static techniques divide and assign work to PEs before application execution, and the assignment does not change afterward. Block, cyclic, and block-cyclic are examples of static scheduling methods [LTS+gu]. In this work, block scheduling where each PE is assigned one block of tasks equals to

$$chunksize_{STATIC} = \left\lceil \frac{N}{P} \right\rceil \tag{2.1}$$

is denoted as STATIC. It incurs minimal scheduling overhead. However, it may results in degraded application performance due to load imbalance for irregular applications or systems. Table A.1 lists the symbols used in the calculation of the chunk size and their description.

**Table 2.1 Scheduling methods**

| | |
|---|---|
| **Work-sharing** | |
| How do they work? | All tasks are assigned as soon as they are created. |
| Challenges | How to distribute tasks to achieve load balance? |
| Load balancing | Proactive, tasks are distributed among PEs in a way that achieves a balanced execution. |
| **Self-scheduling** | |
| How do they work? | Tasks are stored in a central work queue.<br>Chunks of tasks are self-assigned to free and requesting PEs. |
| Challenges | How many tasks to assign at a time? |
| Load balancing | Proactive, tasks are assigned in decreasing chunk sizes to avoid overloading PEs. |
| **Work-stealing** | |
| How do they work? | Tasks are initially distributed to PE work queues.<br>An idle PE steals tasks from overloaded PE queues . |
| Challenges | How to select a victim?<br>How many tasks to steal? |
| Load balancing | Reactive, stealing adjusts the tasks distribution. |

### 2.2.2 Dynamic Loop Scheduling

Dynamic loop self-scheduling (DLS) techniques assign work during execution to free and requesting PEs employing self-scheduling.

DLS techniques have been introduced since 1980s [PPC86] and have been developed and employed by scientific applications to successfully balance their loads on HPC systems, such as heat conduction [BV02], N-Body simulations [BFH95], solar map generation [BWA16], an image denoising model, simulations of a vector functional-coefficient autoregressive (VFCAR) model for multivariate nonlinear time series [CB07], and a computer vision application (PSIA) [EMC17a]. DLS techniques only assume and are applicable to independent tasks or *doall* loops of applications [PPC86; KW85; PK87; FHSF92; BVD03; CB08]. For dependent tasks, several loop transformations, such as loop peeling, loop fission, loop fusion, and loop unrolling, can be used to eliminate loop dependencies [BGS94]. The use of DLS to improve the performance of computationally-intensive scientific applications executing on modern HPC platforms is of increased significance today as system-induced load imbalance is exacerbated due to systems diversity,

**Table 2.2 A summary of loop self-scheduling techniques characteristics**

| Scheduling technique | Category | | | Chunk calculation | | Chunk size | | Use of batches | |
|---|---|---|---|---|---|---|---|---|---|
| | | Dynamic | | | | | | | |
| | Static | Nonadaptive | Adaptive | Deterministic | Probabilistic | Fixed | Variable | Yes | No |
| Static block cyclic (STATIC) | ✓ | | | ✓ | | ✓ | | | ✓ |
| Self-scheduling (SS) [PPC86] | | ✓ | | ✓ | | ✓ | | | ✓ |
| Fixed size chunking (FSC) [KW85] | | ✓ | | | ✓ | ✓ | | | ✓ |
| Modified fixed size chunking (mFSC) [BCS13] | | ✓ | | ✓ | | ✓ | | | ✓ |
| Guided self-scheduling (GSS) [PK87] (GSS) | | ✓ | | ✓ | | | ✓ | | ✓ |
| Trapezoid self-scheduling (TSS) [TN93] | | ✓ | | ✓ | | | ✓ | | ✓ |
| Factoring (FAC) [FHSF92] | | ✓ | | | ✓ | | ✓ | ✓ | |
| Random (RAND) [CIB18] | | ✓ | | | ✓ | | ✓ | | ✓ |
| Weighted factoring (WF) [FHSU+96] | | ✓ | | | ✓ | | ✓ | ✓ | |
| Adaptive weighted factoring (AWF) [BVD03] | | | ✓ | | ✓ | | ✓ | ✓ | |
| Adaptive weighted factoring (AWF-B) [CB08] | | | ✓ | | ✓ | | ✓ | ✓ | |
| Adaptive weighted factoring (AWF-C) [CB08] | | | ✓ | | ✓ | | ✓ | | ✓ |
| Adaptive weighted factoring (AWF-D) [CB08] | | | ✓ | | ✓ | | ✓ | ✓ | |
| Adaptive weighted factoring (AWF-E) [CB08] | | | ✓ | | ✓ | | ✓ | | ✓ |
| Adaptive factoring (AF) [BL00] | | | ✓ | | ✓ | | ✓ | | ✓ |

complexity, increased system size, increased heterogeneity, and massively parallel nature [WLB+16; DBK06].

Table 2.2 [MC18a] summarizes the characteristics of DLS techniques considered in this work. The key difference between various DLS techniques lies in the chunk size calculation, i.e., amount of work (tasks) assigned at a time to a PE per request. DLS techniques are divided into *nonadaptive* and *adaptive* techniques [BC05].

#### 2.2.2.1 Nonadaptive DLS techniques

Nonadaptive DLS techniques calculate chunk sizes based on the probabilistic analysis of loop iteration execution times. They account for load imbalance in scientific applications caused by application characteristics, such as the variation of task execution times. Nonadaptive DLS techniques include self scheduling (SS) [PPC86], fixed-size chunk (FSC) [KW85], modified fixed-sized chunking [BCS13] (mFSC), guided self-scheduling [PK87] (GSS), trapezoid self-scheduling [TN93] (TSS), factoring [FHSF92] (FAC), weighted factoring [FHSU+96] (WF), and random [CIB18] (RAND).

SS assigns a single task at a time per PE request.

$$chunksize_{SS} = 1 \tag{2.2}$$

Therefore, SS achieves the maximum possible load balance at the cost of the highest scheduling overhead among DLS techniques. SS represents one extreme, where STATIC represents the opposite extreme with minimal scheduling overhead and minimal load balancing capacity. FSC assigns tasks in chunks of fixed size. It calculates the optimal chunk size that minimizes scheduling overhead and maximizes the load balance based on the variability among tasks execution times and the scheduling overhead.

$$chunksize_{FSC} = \frac{\sqrt{2}Nh}{\sigma P\sqrt{\log P}} \tag{2.3}$$

Therefore, FSC requires the measurement of $h$ and $\sigma$ before using the technique. mFSC alleviates this burden and assigns a chunk size that results in a number of chunks equal to that of FAC (explained below).

$$tsize = \left\lceil \frac{N}{P} \right\rceil \tag{2.4}$$

$$chunksize_{mFSC} = \frac{0.55 + tsize \log 2}{\log tsize} \tag{2.5}$$

GSS addresses the uneven starting times of PEs and assigns chunks in decreasing sizes.

$$chunksize_{GSS} = \left\lceil \frac{R}{P} \right\rceil \tag{2.6}$$

TSS assigns chunks of decreasing sizes, similar to GSS. However, chunk sizes decrease linearly in TSS, which simplifies the chunk calculation and reduces scheduling overhead. The decreasing sizes help in balancing the PEs loads.

$$\text{First chunk size } f = \frac{N}{2P} \tag{2.7}$$

$$\text{Last chunk size } l = 1 \tag{2.8}$$

$$I = \left\lceil \frac{2N}{f + l} \right\rceil \tag{2.9}$$

$$\text{Decrement between consecutive chunks } D = \frac{f - l}{I - 1} \tag{2.10}$$

$$chunksize_{TSS} = \text{previous chunksize} - D \tag{2.11}$$

FAC assigns chunks in batches to reduce the scheduling overhead. FAC employs probabilistic analysis of application characteristics to calculate batch sizes that maximize the probability of achieving a balanced load execution. The batch size calculation depends

on the mean of tasks execution times, $\mu$, and their standard deviation, $\sigma$. The chunks within a batch are equal in size,namely the chunk size equals to the batch size divided by $P$.

$$chunksize_{FAC} \left\lceil \frac{R_j}{x_j P} \right\rceil \text{, j is the batch number} \tag{2.12}$$

$$b_j = \frac{P}{2\sqrt{R_j}} \times \frac{\sigma}{\mu} \tag{2.13}$$

$$x_0 = 1 + b_0^2 + b_0\sqrt{b_0^2 + 2} \tag{2.14}$$

$$x_j = 2 + b_j^2 + b_j\sqrt{b_j^2 + 4} \tag{2.15}$$

$\mu$ and $\sigma$ need to be calculated before the execution of a loop via profiling. When $\mu$ and $\sigma$ are not available, FAC is practically implemented by assigning half of the remaining loop iterations as a batch, which is equally distributed to PEs on request.

$$chunksize_{FAC} = \left\lceil \frac{R_j}{2P} \right\rceil \tag{2.16}$$

WF is derived from FAC to address heterogeneous PEs. With WF, each PE is assigned a relative weight that is fixed during execution. Each PE is assigned a chunk from the current batch relative to its weight.

$$chunksize_{WF} = w_i \times chunksize_{FAC} \tag{2.17}$$

In this work, the practical implementations of FAC and WF are used. RAND employs the uniform distribution to arrive at a randomly calculated chunk size between an upper and a lower bound [CIB18].

$$\frac{N}{100P} \leq chunksize_{RAND} \leq \frac{N}{2P} \tag{2.18}$$

#### 2.2.2.2 Adaptive DLS techniques

The adaptive DLS techniques measure the performance during execution and adapt their chunk calculations accordingly to address the load imbalance due to systemic characteristics, such as non-uniform memory access (NUMA) delays and perturbations during execution. The adaptive DLS techniques include adaptive weighted factoring [BVD03] (AWF), AWF variants [CB08]: AWF-B, AWF-C, AWF-D, AWF-E, and adaptive factoring [BL00] (AF), among others. AWF adapts the relative PE weights during execution according to their performance. It is designed for time-stepping applications. It measures the performance of PEs during previous time-steps and updates the PEs

relative weights after each time-step to balance the load according to the computing system's present state.

$$chunksize_{AWF} = w_i \times chunksize_{FAC} \tag{2.19}$$

$$\text{Weighted average ratio } \theta_i = \frac{\sum_{j=1}^{s} j \times t_{ij}}{\sum_{j=1}^{s} j \times n_{ij}} \tag{2.20}$$

$$\text{Average weighted average ratio } \bar{\theta} = \frac{\left(\sum_{j=1}^{P} \theta_i\right)}{P} \tag{2.21}$$

$$\text{PE raw weight } \eta_i = \frac{\bar{\theta}}{\theta_i} \tag{2.22}$$

$$\text{Sum of raw weights } \hat{\eta} = \sum_{j=1}^{P} \eta_i \tag{2.23}$$

$$w_i = \frac{\eta_i \times P}{\hat{\eta}} \tag{2.24}$$

AWF-B relieves the time-stepping requirement to learn the PE weights. It learns the PE weights from their performance in previous batches instead of time-steps. It uses timings from earlier chunks to compute the PE weights for the succeeding chunks. This allows for more frequent adaptations since the PE weights can be adjusted while the parallel loop is being executed.

$$\text{Initial batch } = \beta_0 \times N, 0 \leq \beta_0 \leq 1 \tag{2.25}$$

$$1^{st} \text{ chunksize } = \frac{\beta_0 \times N}{P} \tag{2.26}$$

AWF-C is similar to AWF-B. However, the PE weights are updated after the execution of each chunk, instead of batch. AWF-D is similar to AWF-B, where $t_{ij}$ is the scheduling overhead (time taken to assign a chunk of loop iterations) in addition to chunk execution time. This allows accounting for the scheduling overhead in the weight calculation. AWF-E is similar to AWF-C and also takes into account the scheduling overhead, similar to AWF-D.

AF is also based on FAC. It relaxes the assumptions in the FAC that $\mu$ and $\sigma$ are known apriori, and they are the same on all PEs. Therefore, each PE has its own $\mu$ and $\sigma$, i.e., $\mu_i$ and $\sigma_i$ for the $i^{th}$ PE, which are calculated based on the tasks executed locally by this PE. AF dynamically estimates these statistics during execution. When a PE requests a new chunk of work, it reports the performance data of the previously executed chunk.

$$chunksize_{AF} = \frac{D + 2TR - \sqrt{D^2 + 4DTR}}{2\mu_i} \tag{2.27}$$

$$D = \sum_{i=1}^{P} \frac{\sigma_i^2}{\mu_i} \tag{2.28}$$

$$T = \left( \sum_{i=1}^{P} \frac{1}{\mu_i} \right)^{-1} \tag{2.29}$$

# 3

# Perturbations

Unexpected variations in the performance of an HPC system or its components are referred to as perturbations. Perturbations are caused by interference of application-s/processes sharing system resources or unexpected transient or permanent malfunction. Perturbations cause PE reduced computing speed, reduced network bandwidth, longer network latency, or failures (in PEs or network links).

Perturbations are considered another major challenge of scientific applications performance on HPC systems and are expected to be more significant in the future due to fabrication tolerances and thermal concerns [WLB+16]. Due to the large number of components in these systems, perturbations are inevitable. For example, ASC Q system at Los Alamos National Laboratory had on average 26.1 processors failures per week [MHH+05]. Also, the mean time to failure (MTTF) for BlueGene and Titan were found to be 7.9 days [BBC+; DGG+19] and 22.78 hours [Ni16], respectively. Failure rates of a system grow proportional to the number of processors sockets in a system [SG07]. Extrapolating the current failure rates to Exascale systems would result in MTTF of 24 minutes, and if the resiliency of the components is assumed to be improved by a factor of 10, this will result in a failure every 4 hours [SWA+14; BBC+].

## 3.1 Faults, Errors, Failures

A *fault* is a sudden malfunction that occurs in a computing system, such as a bit flip or incorrect control signal. A fault can lead to an *error* when a faulty unit is used in calculations. Errors can be classified as *soft* or *hard* error. *Soft errors* are transient and can not be reproduced. They do not result in permanent hardware damage that persists after the recovery from the error. *Hard errors* are persistent and typically caused by hardware damage. The faulty component needs to be replaced or fixed to recover from hard errors. Errors could be fatal, i.e., fatal perturbations or they could be silent or

dormant errors, such as *silent data corruptions* (SDC) [FME+12b]. Fail-stop failures of computing cores, nodes, or network links that render certain nodes are unreachable are considered and denoted as *fatal perturbations* in the rest of this text.

Besides failures, the availability of PEs to compute or the network latency or bandwidth could be reduced during the execution due to resource sharing or temporary (instantaneous) malfunctions [SWZ+13]. Such errors that disturb system performance but not cause the cease of its operation are referred to as *nonfatal perturbations*.

Maintaining "correct" operation in the presence of failures is denoted as *fault tolerance*, whereas *robustness* denotes the maintenance of certain desired system characteristics despite fluctuations in the behavior of its components or its environment [AMS+04].

## 3.2 Robustness Metrics

Robustness, as defined above, is the ability to maintain a certain performance level in the presence of system fluctuations or perturbations. Perturbations, considered herein, are divided into fatal and nonfatal. Fatal perturbations or failures are events where the system or one of its components cease regular operation, causing the executing applications to fail (if no robust techniques are employed in the application). Examples of such failures are the failure of a compute node or an interconnection link or switch that render certain compute nodes unreachable. Robustness against such failures is denoted as *resilience* [BCCn09; SBC10].

Nonfatal perturbations manifest as performance variability of the system, where the performance of a computing system or one of its components is irregular (typically degraded performance) due to nonfatal errors (see Section 3.5) or other processes or operating system interference. Examples of nonfatal perturbations are reduced PE computing speed, reduced network bandwidth, and longer network latency. Robustness against nonfatal perturbations is denoted as *flexibility* [BCCn09; SBC10].

The Feature Perturbation Impact Analysis (FePIA) [AMS+04] procedure is employed to derive the resilience and flexibility metrics. The FePIA is derived by analyzing the impact of a perturbation factor $\pi$ on a performance feature of interest $\phi$. The metric $\rho$ determines how many folds a performance feature $\phi$ is impacted by a perturbation factor $\pi$. A robustness metric, resilience or flexibility, of an application in a particular execution scenario of 1, denotes the most robust execution.

For resilience, $\rho_{res}(\phi, \pi)$, the performance feature $\phi$ is the parallel execution time of an application $T_{par}$ and $\pi$, the perturbation parameter, is failures, or the number of

failures [SBC10]. Resilience is calculated as

$$\rho_{res}(\phi, \pi) = r_{DLS}/r_{minDLS} \tag{3.1}$$

, where the robustness radius

$$r_{DLS} = T_{par}^{\pi} - T_{par}^{orig} \tag{3.2}$$

and

$$r_{minDLS} = MIN(r_{DLS}), \forall DLS \in [STATIC, SS, FSC, mFSC, FAC, ...] \tag{3.3}$$

for a certain perturbation parameter $\pi$. $T_{par}^{\pi}$ is the parallel execution time under perturbation $\pi$ and $T_{par}^{orig}$ is the parallel execution time in the perturbation free execution.

Similarly, flexibility, $\rho_{flex}(\phi, \pi)$, is calculated as

$$\rho_{flex}(\phi, \pi) = r_{DLS}/r_{minDLS} \tag{3.4}$$

where $\phi$ is the parallel execution time $T_{par}$ and $\pi$ is the perturbation parameter, i.e., system performance variability, such as PE perturbations, prolonged latency perturbations, and reduced network bandwidth perturbations. Resilience and flexibility have been used in literature to analyze and evaluate the robustness of various nonadaptive and adaptive DLS techniques on homogeneous and heterogeneous HPC systems [SCB10; SSB+12; BBC14; SBS+13a; SBC15; BBC13].

## 3.3 Perturbations in HPC Systems

HPC systems are large and complex systems that are built with quality components. However, due to the large number of components in these systems, failures are inevitable. Terascale systems with thousands of sockets experienced failures every 8 - 12 hours, corresponding to 125 - 83 million failures in time (FIT) [BBC+]. The most significant indicator of failure rate was socket count, and hardware was the most dominant source of failures. The sheer number of DRAM chips in *BlueGene* systems was the most significant contributing factor to failures [BBC+]. The mean time between failures (MTTF) for BlueGene was found to be 7.9 days [BBC+]. Also, for *ASC Q* system at Los Alamos National Laboratory had on average 26.1 processors failures per week [MHH+05].

At the Petascale, studying 160,000 entries of failures recorded for *Titan* system at Oak Ridge Leadership Computing Facility (OLCF), which has a peak performance of 27 PetaFLOP/s, revealed that its mean time between failures (MTBF) is 22.78 hours [MNJ+15; Max08]. In *Blue Waters* system, 52% of failures in 261 days were hardware-related and needed small downtime. 99.3% of hardware failures are contained

in a single blade, which is consistent with failures on *Titan*, i.e., failures have spatial locality. Extrapolating the current failure rates to Exascale systems would result in MTTF of 24 minutes, and if the resiliency of the components is assumed to be improved by a factor of 10, this will result in a failure every 4 hours [BBC+].

A single bit flip in memory can be detected with cyclic redundancy check (CRC) and even mitigated with error correction code (ECC). Double bit flips, however, force an instant reboot after detection since ECC cannot correct such faults. Double-bit errors were observed once every 24 hours in *Jaguar* 360 TB memory [Rus16]. On a Cray XT5 system at Oak Ridge National Laboratory, double bit flips occur at a rate of one per day for 75,000 memory modules [Gei11; FME+12a]. Single bit flips in the processor core remain undetected as only caches feature ECC while register files or even arithmetic logic units (ALUs) typically do not. Significant SDC rates were also reported for *BlueGene/L* unprotected L1 cache, which explains the ECC in L1 caches of *BlueGene/P* [BM09]. In a recent study of memory errors in over two years of large scale Google clusters [SPW11], it was found that 8% of memory modules per year are affected.

Network failures are also very critical to large scale HPC systems. Small applications with many-to-few communication pattern may result in network congestion events [KGP+18]. Network errors on *Titan* were found to have a spacial correlation and are unevenly distributed. The study of network failures and their recovery procedures on *Blue Waters* system showed that failures during recovery caused 28 out of 101 system-wide outages [JFDM+17].

The study of failures of *Jaguar* and *Titan* systems between years 2008 and 2015 shows that file system errors are dominant and of equal importance to memory errors [GPE+17]. Large scale file system is the foremost source of failures, and *when file system fails, checkpointing to file system is not very effective [MNJ+15].*

## 3.4 Lessons Learned form the Analysis of System Logs

Failure rates of a system grow proportional to the number of processors sockets in a system [SG07]. The study of failures in large scale systems revealed that there is a correlation between the failure rate of a system and the type and intensity of the workload execution on it [SG10]. Also, failures exhibit temporal correlation and are spatially correlated as well, especially for network root-caused failures. Failures are not uniformly distributed across the system and are more likely to reoccur in the neighbor of previous failures [GPE+17; DGS+17; PLS+17]. Memory errors are found to be proportional to utilization and exhibit spacial locality [SPW11; ESS13]. Temperature does not seem to affect failures. Surprisingly, aging has a strong influence on increasing error rates in

memory modules.

Power outages and network failures have a powerful effect on subsequent failures; 30% − 50% of nodes experience at least one failure in the following week (temporal locality) [ESS13]. Moreover, temporal locality of different failure types are significantly different, but similar across systems [GPE+17]. Cosmic rays do not affect DRAM errors, which might indicate the efficiency of ECC. However, CPU errors are significantly influenced by cosmic rays [ESS13]. Interestingly, faults in SRAM (used for cache memory) increase with altitude of the system, which confirms that memory is venerable to faults due to high-energy particle strikes [SDB+15].

It was found that the time between failure at individual nodes, as well as at an entire system, is fit well by gamma or Weibull distributions with decreasing hazard rate (Weibull shape parameter of $0.7-0.8$) [SG10]. Both exponential distribution and Weibull distributions with shape parameter 0.82 fit the MTBF of *Titan* [MNJ+15]. Measurements from different sensors can point to an impending failure [LGZ+10; GCS+12; FX07] as well. Therefore, these failure distributions and sensor measurements could be used to predict failures and proactively tolerate failures.

## 3.5 Impact of Nonfatal Perturbations on Scientific Applications Performance

Fatal errors cause the application failure or restart from a checkpoint. Nonfatal errors do not cause application or system failures. However, they perturb the applications' performance. Corrected memory errors degraded the performance of SPEC CPU2006 by 2.5 times on average, due to the interference of hardware interrupts and error reporting stack [GSG+16].

An approach to identify such perturbed executions was proposed where the application performance is analyzed between collective MPI calls and the execution times of such segments are classified to identify intrinsic and extrinsic application performance variations [SMW18]. Therefore, the amount of system noise can be measured based on the measured extrinsic performance variations in application segments.

The impact of nonfatal perturbations on applications performance was studied on 18,688 nodes of *Titan* at OLCF for 13 months (Dec. 2015 - Feb. 2017) [AE18]. 68,000 applications out of 2 million applications recorded reliability, availability and serviceability (RAS) events during their execution. Applications performance was examined during several executions where RAS events are recorded (a nonfatal perturbation occurred) and not recorded (error-free execution). Performance results in Figure 3.1 show that nonfatal perturbations severely impact applications' performance.

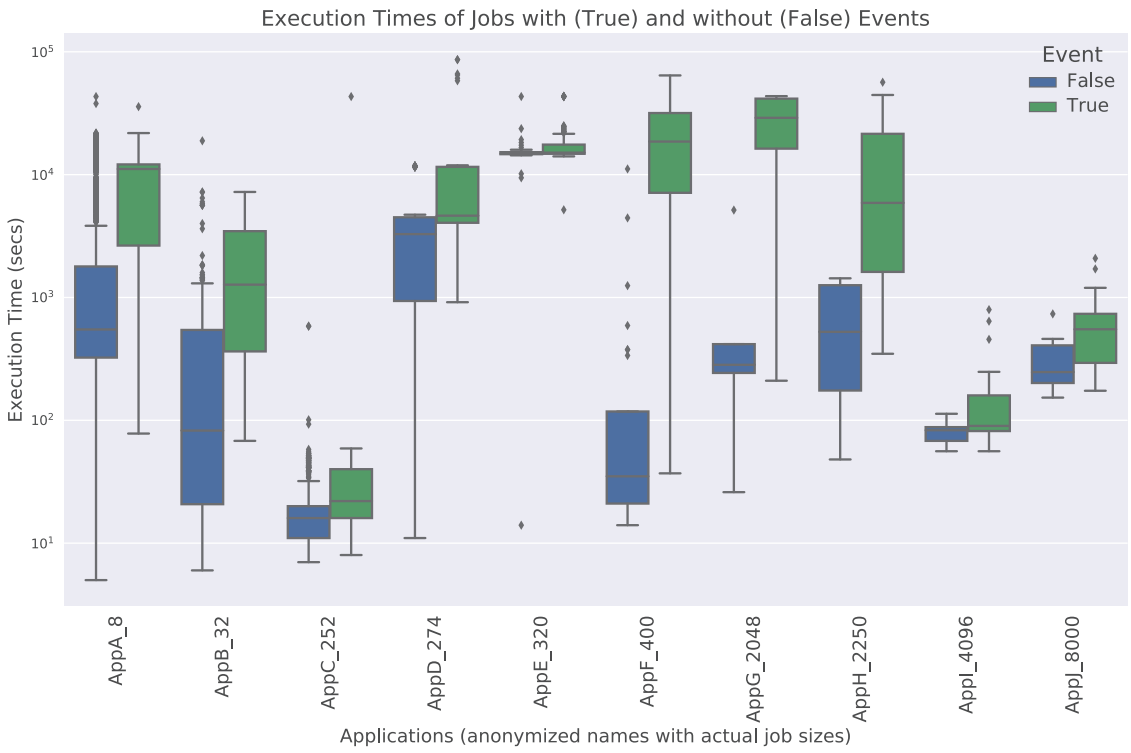



**Figure 3.1 Impact of nonfatal perturbations on applications' performance on *Titan*.** Comparison between the application performance with and without the occurrence of RAS events. Nonfatal errors perturb applications performance and degrade their performance significantly [AE18].

The impact of these perturbations can also be more significant with heavy workloads. For instance, the execution time variability of acceptance tests of the *Summit* system was found to be over the desired threshold when the system was fully loaded [VLWB+19]. One core per socket was dedicated to system services to reduce such variability in the execution time. Also, contention on file system due to checkpointing could slow down applications performance up to five times [SKN+19; KSN+14; KNM+13].

## 3.6 Discussion

We showed the most common reported failures on large scale HPC systems. Interestingly, not only fatal system errors affect applications performance, but also nonfatal errors that systems recover from during execution perturb applications and degrade their performance. However prevalent, very little work considered uncertain communication time and no work considered perturbations in the interconnection bandwidth and latency and their effect on applications' performance. This defines the gap that will be

addressed in the next chapters, in addition to fatal and nonfatal perturbations.

# 4

# Performance Simulation

Due to the large number of DLS techniques and their various characteristics, identifying the best choices among the available DLS techniques for a given application requires intensive assessment and a large number of exploratory native experiments. This significant amount of experiments may not always be feasible or practical due to their associated time and costs. A theoretical model that can be used to predict the scheduling technique that yields the best performance for a given problem and system has not yet been identified [MEC+19]. Simulation mitigates such costs and, therefore, is more appropriate to study the performance for optimization [STL+15]. Also, simulation enables experiments on computing systems that do not exist (from the past and no longer exist or an anticipated future system). Studying application performance via simulation provides the control needed to understand the effect of each factor (e.g., network latency, computing speed, and the number of tasks) that contributes to the performance in isolation of other factors. Simulation allows the study of application performance in controlled and reproducible environments [STL+15]. Realistic simulation predictions lead to the improvement of applications' performance in native executions.

## 4.1 SimGrid Simulation Toolkit

Our proposed realistic simulation approach is exemplified with the SimGrid [CGL+14] toolkit, which is used throughout this doctoral thesis. However, the proposed realistic simulation approach can be applied to simulate applications in other simulators that provide functionality and architecture similar to the SimGrid.

SimGrid (hereafter, SG) is a scientific simulation framework for the study of the behavior of large-scale distributed computing systems, such as the Grid, the Cloud, and peer-to-peer (P2P) systems. It provides application programming interfaces (APIs) to simulate various distributed computing systems. Various studies have used SG to study

the performance of applications with DLS techniques in different scenarios [BBC+17; SBS+13b; SMS+14].

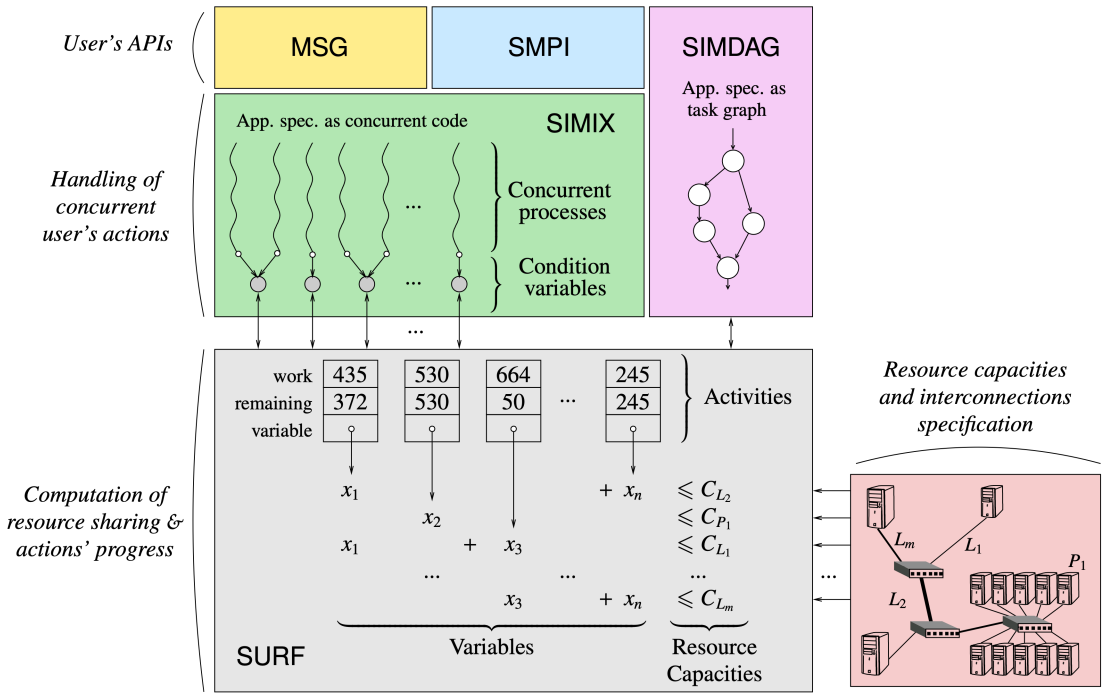



**Figure 4.1 SimGrid architecture and components [CGL+14].**

The architecture of SG is shown in Figure 4.1. The performance simulation is performed by the event-based simulation engine *SURF* (gray area). It uses a fast and straightforward CPU computation model, where a task execution time is calculated as the amount of work in the task represented in the number of floating-point operations (FLOP) divided by the computing speed of the PE executing this task, represented as FLOP/s. For the network model, SG uses a flow-level approach that approximates the behavior of TCP networks, including bandwidth sharing and contention. SimGrid network models have been verified to approximate the network behavior with good accuracy [VL09]. These properties render it well suited for the study of large-scale, computationally-intensive distributed scientific applications.

*SURF* interacts with both the computing system representation (light red area) and the user interfaces (top area) to read in the computing system and application characteristics, respectively. SG provides four different user APIs for different simulation purposes.

1. MetaSimGrid (SG-MSG)
2. SimDag (SG-SD)
3. SG-SMPI

**4.** SG-S4U

MetaSimGrid (SG-MSG) is for representing parallel applications, where an application is represented as a set of parallel communicating processes. Processes represented by SG-MSG can implement a master-worker execution model. Also, SG-MSG provides a deployment file to map processes to PEs. Listing 4.1 shows a sample SG-MSG simulation of 10 tasks executed by 4 workers. The master only distributes the work in this example.

SimDag (SG-SD) is for representing applications as tasks or task graphs using directed acyclic graphs (DAGs). A task can be a computation task or a communication task to represent computation and communication performed by an application, respectively. SG-SD supports the creation of dependencies between tasks (computation or communication) to represent edges in a DAG. For the mapping of tasks to PEs, SG-SD provides a scheduling function to schedule a task on a PE. Scheduled tasks are executed as soon as all their dependencies are satisfied. Listing 4.2 shows a sample SG-SD simulation that creates two computation tasks and one communication task and executes these tasks on two PEs.

The SG-SMPI interface provides the functionality for the simulation of programs written using the message passing interface (MPI) and targets developers interested in the simulation and debugging of their parallel MPI codes. SG-SMPI maps every MPI rank of an application onto a lightweight SG thread. Simulation threads are run sequentially by the SG simulation engine, *SURF*. Each time a simulation thread makes an MPI call, SG-SMPI calculates the time that was spent computing (isolated from the other threads) since the previous MPI call. This time is scaled up or down depending on the speed of the simulated machine relative to the simulation machine and is injected into the simulator.

The newly introduced SG-S4U interface (under development) currently supports most of the functionality of the SG-MSG interface and also will incorporate the functionality of the SG-SD interface over time.

SG uses an XML file of a special format, denoted as `platform file` to describe the computing system characteristics. A PE is represented by a host in the SG `platform file`. A host contains a certain number of cores, and all cores use a fixed compute speed defined in FLOP/s. Hosts can be connected by various links to create the required network topology. A network link is characterized by its bandwidth, latency, direction, and *sharing policy*. If a link is shared, by specifying keyword "SHARED" in the link *sharing policy*, then the link bandwidth will be divided among simultaneously communicating hosts. If it is not shared, i.e., "FATPIPE" as the link *sharing policy*, then SG will create automatically a separate link for every pair of communicating hosts.

Listing 4.1: Sample SG-MSG simulation with 4 workers and 10 tasks

```
#include "simgrid/msg.h"
/*The master process */
static int master(int argc, char *argv[])
{
  for (int i = 0; i < 10; i++) {
    char mailbox[80];
    char tname[80];
    /*1000 FLOPS of computations and 0 bytes of communicaion*/
    comp = 1000;comm = 0;
    snprintf(mailbox,79, "worker-%ld", i % 4);
    snprintf(tname,79, "Task_%d", i);
    msg_task_t t = MSG_task_create(tname, comp, comm, NULL);
    MSG_task_send(t, mailbox);
  }
  for (int i = 0; i < 4; i++) {
    char mailbox[80];
    snprintf(mailbox,79, "worker-%ld", i % 4);
    msg_task_t finalize = MSG_task_create("finalize", 0, 0, 0);
    MSG_task_send(finalize, mailbox);
  }
  return 0;
}
/* Worker processes */
static int worker(int argc, char *argv[])
{
  char mailbox[80];
  long id= xbt_str_parse_int(argv[1], "Invalid argument %s");
  snprintf(mailbox,79, "worker-%ld", id);
  /* Wait in an infinite loop for tasks sent by the master */
  while (1) {
    msg_task_t task = NULL;
    int res = MSG_task_receive(&task, mailbox);
    if (strcmp(MSG_task_get_name(task), "finalize") == 0) {
      MSG_task_destroy(task);
      break;
    }
    MSG_task_execute(task);
    MSG_task_destroy(task);
  }
  return 0;
}
int main(int argc, char *argv[])
{
  MSG_init(&argc, argv);
  MSG_create_environment(argv[1]);
  MSG_function_register("master", master);
  MSG_function_register("worker", worker);
  MSG_launch_application(argv[2]);
  msg_error_t res = MSG_main();
  return 0;
}
```

Listing 4.2: Sample SG-SD code

```
#include "simgrid/simdag.h"
#include "xbt/log.h"

int main(int argc, char **argv)
{
  SD_task_t task;
  SD_init(&argc, argv);
  SD_create_environment(argv[1]);
  sg_host_t *hosts = sg_host_list();

  /* creation of some typed tasks and their dependencies */
  SD_task_t comp1 = SD_task_create_comp_seq("T1", NULL, 1e9);
  SD_task_t comm1 = SD_task_create_comm_e2e("comm", NULL, 1e7);
  SD_task_t comp2 = SD_task_create_comp_seq("T2.", NULL, 1e9);

  SD_task_dependency_add(NULL, NULL, comp1, comm1);
  SD_task_dependency_add(NULL, NULL, comm1, comp2);


  SD_task_schedulel(comm1, 1, hosts[0]);
  SD_task_schedulel(comp2, 1, hosts[1]);

  xbt_dynar_t ctasks = xbt_dynar_new(sizeof(SD_task_t), NULL);
  SD_simulate_with_update(-1.0, ctasks);

  xbt_dynar_foreach(ctasks, ctr, task)
    SD_task_destroy(task);
  xbt_dynar_free_container(&ctasks);
  xbt_free(hosts);
  return 0;
}
```

Listing 4.3: Sample SG `platform file`

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM
"http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
<platform version="4.1">
  <zone  id="AS0"  routing="Full">

    <host id="nodeo01" core="10" speed="1Gf"/>
    <host id="node02"  core="16" speed="0.5Gf"/>
    <link id="1" bandwidth="10kBps" latency="10ms"
      sharing_policy="SHARED"/>
    <route src="nodeo01" dst="node02" direction="NONE" >
      <link_ctn id="1"/>
    </route>
  </zone>
</platform>
```

# 5

# State of Practice

## 5.1 Load Balancing

There is no standard load balancing method across compute nodes at the time of writing for applications parallelized with message passing interface (MPI), the most common parallelization approach used in scientific applications. Most applications, parallelized with MPI, use a static division of the work (by equally dividing the domain over the processes). To find a domain decomposition that balances the load, space-filling curves and curve cutting heuristics are employed [LN18; HKR+12]. For load-balancing these applications, the domain decomposition is adjusted at certain time intervals, to re-balance the load as the application execution evolves. An overview of the most common and successful domain decomposition methods for load balancing, such as space-filing curves, recursive coordinate bisection [BB87], clustering, and hypergraph methods is presented [DBK06]. However, these methods are not suitable to address load imbalance caused by systemic characteristics, such as NUMA effects, system interference, or perturbations.

Other widely used programming models, such as Charm++ [KK93], offers certain load balancing heuristics across nodes and is used by scientific applications, such as Jacobi3D, LeanMD, and ChanGa [Ni16; KAB+12; PBW+05; MWZ+15]. Charm++ uses C++ objects called chares that contain work and data. The domain is over-decomposed into chares, where the number of chares is more than the number of PEs in the systems. These objects are migrated between nodes to achieve a balanced execution. MPI applications can also benefit from Charm++ load balancing by compiling and running through adaptive MPI (AMPI) [HLK03]. However, applications using AMPI can not use global variables, which constrains the number of possible applications. Also, multithreaded MPI applications need to be modified as chares are single-threaded entities [WLB+16]. Load balancing methods that rely on predictable applications and computing systems hard-

ware characteristics can not accommodate unpredictable system-induced load imbalance sources, such as data access latencies and operating system interference [DBK06]. Therefore, dynamic load balancing via DLS is essential for improved applications' performance on future HPC systems.

For load balancing within a compute node (in shared memory systems), OpenMP scheduling is the most widely-used method. OpenMP currently supports three scheduling options, static, dynamic, and guided, which are equivalent to STATIC, SS, and GSS, respectively (see Section 2.2). Additionally, certain OpenMP implementations, such as LLVM[1] [LA04] runtime library, supports TSS as a fourth option for scheduling. However, as the number of PEs per node is increasing, with possible heterogeneity and irregular performance [WLB+16], the limited options of scheduling offered by OpenMP may not be suitable to achieve load balance on future HPC systems.

### 5.1.1 Related Work on Two-level Scheduling

Hierarchical domain decomposition with space-filling curves was used to balance the execution of an atmospheric cloud model [LN18]. An exact algorithm was used to cut the space-filling curve, and the cut parts were assigned to PEs using a heuristic. The work division and assignment were performed in two stages to simplify the scheduling problem. However, only scheduling at the process level was considered. Adapting the binding of threads to cores in MPI+OpenMP applications during execution using Quo [Gut18] was introduced. It addresses the fork/join nature of OpenMP threads during a lifetime of an application. Quo assigns more cores to overloaded PEs within an application, preserving data locality. However, an overloaded thread or process may still cause load imbalance at the thread or the process level. ChaNGa [MWZ+15] employs over-decomposition and migration of chares for dynamic load balancing. Its runtime collects load information, and chares are migrated between nodes to achieve load balance [MJZ+12]. In addition to the task migration to balance the load across nodes, OpenMP tasks were used to balance the load within nodes [BMW+18]. OpenMP tasks are created to help overloaded chares in applications, such as Lassen, Kripke, and ChaNGa. Self-scheduling for MPI+OpenMP applications using hierarchical loop scheduling [WYL+12] (HLS) was introduced. In HLS, a master-worker was employed. Upon work request, the master assigns a number of chunks equal to the number of OpenMP worker threads in the node that initiated the request. However, this is different from two-level load balancing, where worker MPI ranks are assigned a single chunk of work per request by the master, which is further distributed among the threads within this rank.

[1] https://llvm.org/

Load balancing solutions based on domain decomposition can not adapt to accommodate unpredictable system variations. Language-specific solutions, such as Charm++, can not easily be ported to other programming languages and models. In the next chapters, we investigate the two-level dynamic load balancing via self-scheduling. We analyze the interplay between load balancing at the thread level and process level and investigate the impact of dynamic load balancing using both dynamic nonadaptive and adaptive self-scheduling techniques.

## 5.2 Fault Tolerance

Several methods are developed and are used to achieve robust application execution in the presence of fatal system failures and SDCs. We present here an overview of the most successful methods with a focus on the robust scheduling (our main interest), namely as *algorithm-based fault tolerance (ABFT), checkpointing, replication, and robust scheduling.*

### 5.2.1 Algorithm-based Fault Tolerance

ABFT is a successful and widely used mechanism for fault tolerance. It was initially proposed for linear algebra kernels due to its low overhead. ABFT exploits algorithmic features within a mathematical kernel in the application to encode redundant computations that are invariant checksums [CC19]. These checksums can be used to detect and correct an error. Examples of problems where ABFT is applicable are matrix multiplication [HA84], dense matrix factorization [DBB+12], sparse matrix conjugate gradient [SSR12], iterative matrix-vector operations [TSK+16], multigrid algorithm iterative solution of Poisson equations [MB03], and stencil computations [CC19]. ABFT has also been used in highly parallel distributed matrix-matrix multiplication [BDD+09] and HPL benchmark [WYC+11]. ABFT is also used in conjunction with checkpointing to improve applications' robustness to failures [Che13]. However, ABFT is only applicable for certain types of applications and is more a problem specific fault tolerance solution rather than a generic solution that is applicable for scientific applications in general.

### 5.2.2 Checkpointing

Checkpointing is the de facto method for the rollback recovery from fatal perturbations [CL85; EAW+02]. Checkpointing is the saving of a snapshot of the application state into storage. In case of the occurrence of a failure, the application state is recovered/reconstructed from the most recent saved checkpoint. The optimal checkpointing frequency is identified and can be calculated based on failure rates [Dal06;

You74]. Several varieties of checkpointing were investigated on the level of checkpoint, such as operating system level [HD06], compiler level [KBP95; LF95; PBK95], and user-level API [BGTK+11; HSM+07]. Checkpoints can be stored to disks, solid-state drives (SSD), memory, and non-volatile memory (NVM). Checkpoints of parallel distributed applications can be coordinated or uncoordinated [DHR15]. Disk checkpoints has been used to tolerate fail-stop errors, where in-memory checkpoints are used to handle SDCs [BCR+16]. Dynamic node replacement could help with the increasing failure rates of large scale HPC systems [PNW18]. However, when the file system fails, checkpointing to file system is not very effective [MNJ+15]. Checkpoints can be performed on a single level of software (thread, process, application, system) or in a hierarchical approach on more than one level. The cost of single-level checkpoints grows with error probability and can be prohibitive for Exascale systems [BBB+14; FSLI+11; PLP98].

### 5.2.3 Replication

Replication methods are robust against fail-stop failures and SDCs [LV62]. Replication is understood as the re-execution of some or all the processes or computations of an application. Replicated process or computations can be executed in parallel with the original copy, by using more hardware resources (system redundancy) or executed after the original copy on the same resources allocated to the original copy (time redundancy). Redundancy at the MPI level with RedMPI was able to detect faults in the system that was not reported by normal MPI [FME+12a].

The main drawback of redundancy is efficiency. However, redundancy may increase the overall efficiency when failure rates are sufficiently high [Ni16]. Partial replication of selected most vulnerable processes and data was introduced to detect and correct SDCs and reduce the overhead by 70% than that of duplication (re-execution of all processes twice) [BBGD+17]. Dynamic Double Modular Redundancy (DDMR) is where component failures are recovered by either using a spare component in the case of a hard error or rebooting the failed one in the case of a soft error [EOS09]. This technique lowers the MTTR of components significantly by a factor of $1,000 - 10,000$ [EOS09]. Simulations showed that replication outperforms traditional checkpointing/recovery methods on systems with more than $20,000$ sockets [FSLI+11].

### 5.2.4 Robust Scheduling

The flexibility of nonadaptive DLS techniques on distributed heterogeneous HPC systems was studied [GGA+17]. Machine learning was used to analyze and create a portfolio of DLS techniques flexibility, and the most robust DLS technique was intelligently

selected to improve applications performance [SMS+14]. However, none of the work above considered sources of perturbations other than the variability of PEs' availabilities, such as unpredictable network latency or bandwidth. RUMR [YC03] was introduced as a robust scheduling method against unpredictable task execution time and unpredictable communication time. Multi-objective evolutionary algorithm and a robust version of HEFT [THW02] were introduced for the robust scheduling of tasks with uncertain computation and communication time [CJ10]. However, static scheduling methods such as robust HEFT or evolutionary algorithm optimization can not adapt to unpredictable perturbations and failures that might occur during execution. Dynamic selection of the most robust DLS technique using reinforced learning was introduced [BBC+17]. The robustness of DLS techniques is learned during previous time-steps in time-stepping scientific applications, and the most robust DLS technique is selected. DLS selection methods above are reactive rather than proactive and depend on detecting perturbations during execution based on the measured performance. Also, non-preemptive execution of scheduled tasks result in poor performance in specific instants due to frequent change of the selected DLS.

A fault-tolerant approach for DLS was introduced and studied using simulation [SBC15], where failed tasks were rescheduled dynamically to working PEs. Different numbers of failed PEs were simulated that represent 12.5%, 25%, and 50% of the total PEs in the system, and the resilience of different DLS techniques were measured based on the number of tasks that needed to be rescheduled in each case. The above work introduced fault tolerance to the DLS techniques. However, it is a reactive approach and depends on detecting a PE failure and then reacting by rescheduling the failed tasks on working PEs. Moreover, the failure detection method was not explicitly clear in that work, as it was only studied in a simulation.

Fault tolerant self-scheduling [WNC+12] (FTSS) was introduced for shared memory systems. FTSS incorporates work-stealing with self-scheduling for fault tolerance. An idle thread will steal work from other loaded/failed thread after all loop iterations are already scheduled. However, work-stealing and victim selection depend on failure detection in FTSS, which was not discussed in this related work.

Fault-tolerant work-stealing [WJS+13] (FTWS) was presented for distributed memory systems. Tasks are duplicated and executed on two different PEs to detect transient and permanent PE failures. Failed tasks are pushed to the faulty task queue, where PEs executes their tasks as soon as they finish their original work queue. An approach was proposed for enhancing the flexibility of scheduling of a bag of tasks on computing grids [DSCB03]. Only perturbations in the computing resources and variations in task sizes were considered, and the proposed method was evaluated in simulation. Similarly,

restarting-failed-tasks-based robust scheduling approaches can also be configured for Apache Spark and Hadoop YARN [VMD+13].

## 5.3 Simulation

A combination of simulation and trace replay was used to guide the choice of the scheduling technique and the granularity of problem decomposition for a geophysics application to tune its performance [KTMSL+17]. SG-SMPI was used to generate a time-independent trace (TiT) of the application with the finest problem decomposition. After that, this trace was modified to represent different granularities of problem decomposition. Traces that represent different decompositions were replayed with different scheduling techniques to identify the decomposition granularity and scheduling technique combination that would result in improved application performance. The scheduling techniques were extracted from the Charm++ [KK93] runtime to be used in the simulation. However, the process of trace modification to represent different decompositions is complex, limits the number of explored decompositions, and may result in inaccurate simulation results.

The compiler-assisted native application source code transformation to a code skeleton suitable for structural simulation toolkit [RBB+12] (SST) was introduced [WKK+18]. Specific pragmas are inserted in the source code to simulate computations as certain delays, eliminate large unnecessary memory allocations in simulation, and handle global variables correctly. This approach was focused on the simulation for the study of communications and networks in large computing systems. Therefore, the variability of task execution times was not considered explicitly.

StarPU [ATN+11] was ported to SG-MSG for the study of the scheduling of task graphs on heterogeneous CPU/GPU systems. Tasks execution times were estimated based on the average execution time benchmarked by StarPU. Both average task execution time and the generation of pseudo-random numbers with the same average as task execution time were explored. However, depending on time measurements, may not be adequate for fine-grained tasks. Also, porting the StarPU runtime to a simulator interface could be challenging and requires much effort.

Also, SG-SMPI was used to simulate and predict the performance of the high performance linpack (HPL)[2] benchmark on large scale HPC systems [CLH19]. Performance models of computational kernels inside the HPL, such as the `dgemm` and `dtrsm`, are used to skip the real computation in SG-SMPI and speed up the simulation.

[2] https://www.netlib.org/benchmark/hpl/index.html

The Monte-Carlo method [MU49] was used to improve the simulation of workloads in cloud computing [BGG18]. Variability in cloud computing systems was quantified and added to task execution times as a probability, to capture the variable application execution times in simulations. The simulation was repeated 500 times, each with different seeds to obtain a similar effect of the dynamic native execution on the clouds. However, variable application execution time has two components: (1) variability in a task execution time due to application characteristics or system characteristics such as non-uniform memory access; (2) the variability that stems from the computing system resources being perturbed by operating system interference, other applications that share resources, or transient malfunctions. Considering both application performance variability components is essential to obtain realistic simulation results.

Part II

# Simulation and Verification

# 6

# Simulation of Applications Performance on High Performance Computing Systems

Simulation is considered the third pillar of science after theory and experimentation. It mitigates substantial costs of exploratory experiments and is more appropriate to study applications' performance for optimization [STL+15]. Simulation enables experimentation on computing systems that do not exist (from the past and no longer exist or an anticipated future system). Also, it allows the study of application performance in controlled and reproducible environments [STL+15]. Realistic predictions based on accurate simulations drive the optimization of native application performance and help pinpoint significant performance issues.

In this chapter, we propose a realistic simulation method for the fast and accurate performance simulation of scientific applications on HPC systems (see the part in color in Figure 6.1). We introduce a generic and systematic method for realistic performance simulations. *Realistic simulation results lead to a similar analysis and conclusions to the analysis of the native results.* Various factors that affect applications' performance on HPC systems are discussed. A set of guidelines and methods to represent applications and computing characteristics is introduced and discussed. We propose and present two simulation approaches, SG-SD and SMPI+MSG, for the simulation of task-based parallel applications and MPI-based parallel applications, respectively. The proposed simulation approaches presented below are used in the next chapters in predicting the performance of scientific applications on past and present computing systems under various execution scenarios.

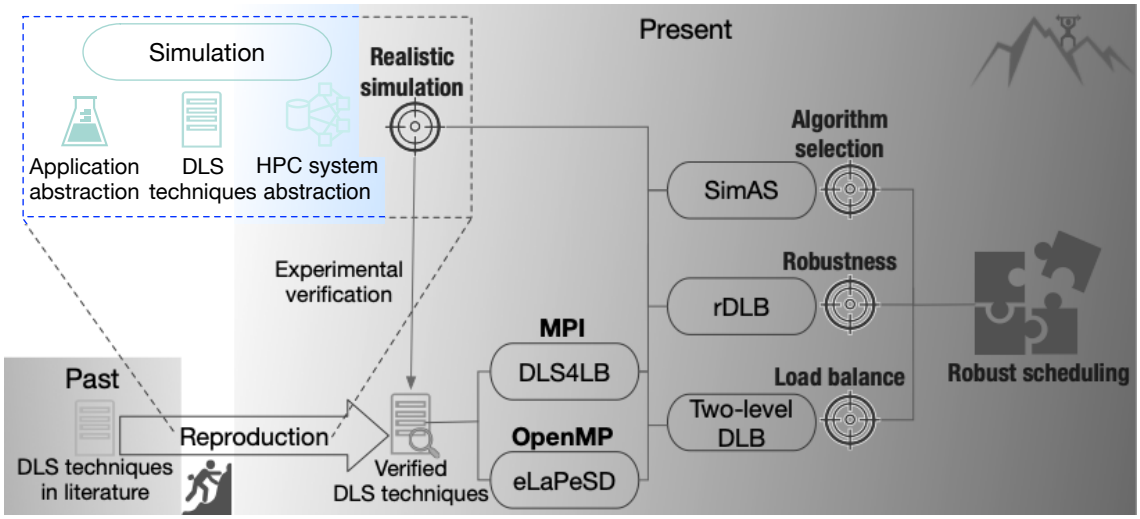



**Figure 6.1 Illustration of the focus in this chapter (in colors) as part of the overall approach (in grayscale).** We present methods to capture and describe applications, DLS, and HPC systems characteristics for the realistic simulation of application performance with DLS.

## 6.1 A Method for Realistic Simulations

*A realistic performance simulation denotes that conclusions drawn from simulative performance results are similar to those drawn from native performance results.* The close agreement between both conclusions does not necessitate the close agreement between native and simulative application execution times. For the study of dynamic load balancing and task self-scheduling, the performance of different scheduling techniques relative to others should be preserved between native and simulative experiments. This would suffice to conclude the similar performance characteristics of DLS techniques between native and simulative experiments.

Realistic simulation is challenging due to the dynamic interactions between the three main components that affect the performance;

1. Application characteristics
2. Dynamic load balancing
3. Computing system.

Figure 6.2 shows these three main performance components and introduces the realistic simulation approach. Each component should be separately represented and verified to achieve realistic simulations. The details of representing the application and computing system characteristics are provided next. The implementation of DLS techniques is described in depth in Chapter 9.

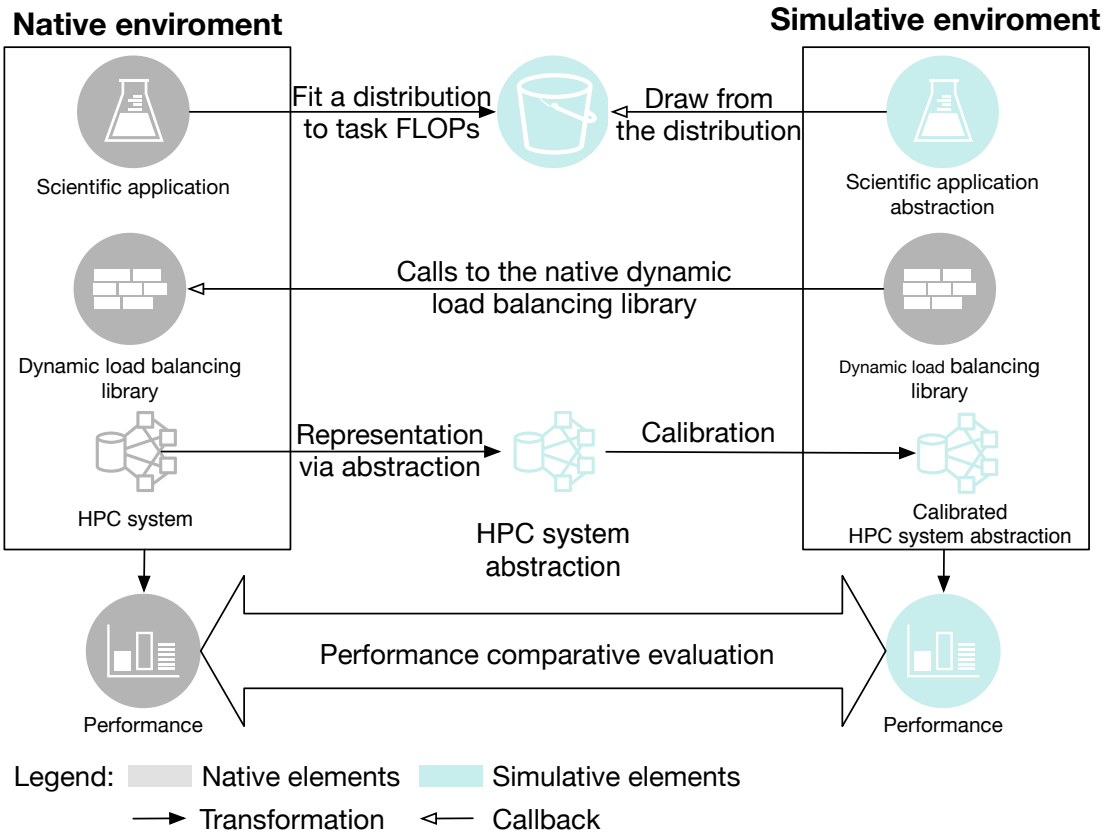



**Figure 6.2** **Illustration of the proposed realistic simulation method.** Scientific application and computing system characteristics are abstracted for use in simulation. Minimizing the changes needed to simulate an application promotes usability and simulation accuracy. A single scheduling library is used, which is called both by the native and simulative executions.

## 6.2 Representing Applications Characteristics

Two important aspects need to be specified to enable representation via abstraction of an application:

1. The main application flow, i.e., initializations, branches, and communications between its parallel processes or threads

2. The computational effort associated with each task

For simple applications with one or two large loops or parallel blocks of tasks that dominate its performance, inspecting the application code may be sufficient to understand the program flow. If this is insufficient, tracing the application execution could reveal the main computation and communication blocks in the application. In particular, the SG-SMPI simulation produces a special type of text-based execution trace called time independent trace (TiT) [DMS12]. The TiT contains a trace of the application execu-

tion as a series of computation and communication events, with their corresponding amounts specified in terms of floating-point operations (FLOP) and bytes, respectively. The SG-SMPI measures the execution time of the computation tasks and multiplies this time by the platform computation speed to estimate the FLOP count per task. Therefore, the TiT can be used to understand the application flow and to represent the application in simulation. Figure 6.3 shows a sample annotated TiT to describe the different fields included in a TiT produced by SG-SMPI.

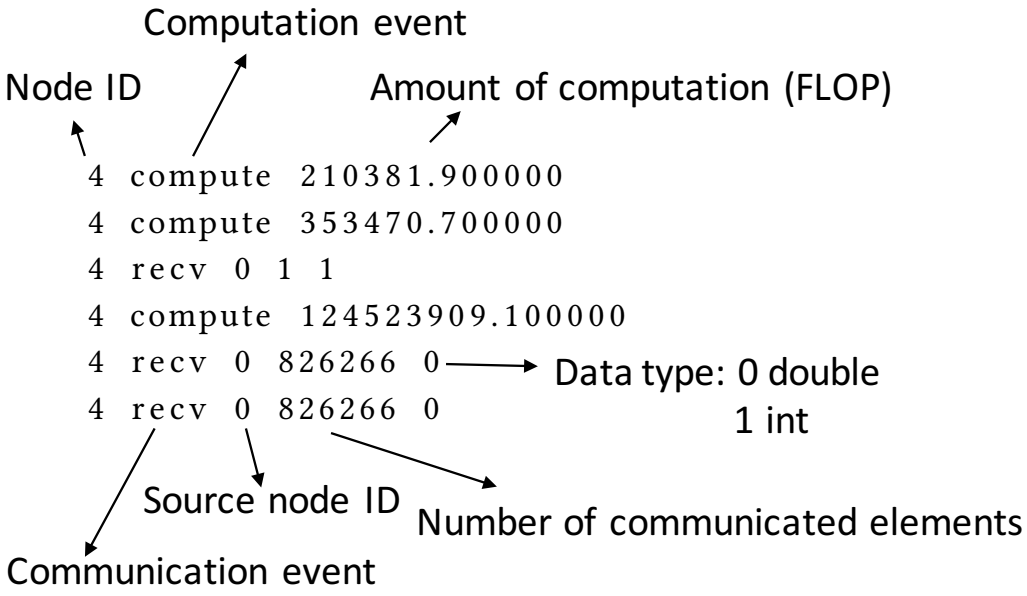



**Figure 6.3 A sample TiT produced by the SG-SMPI and explanation of its different fields [MEC17a].**

Time measurement of task execution times or the FLOP counts can be used to obtain the amount of work per task. The measurement of short task execution times can be a source of measurement inaccuracies as such measurements are affected by the measurement overhead, known as the probing effect. Also, the execution time per task may not be constant between different executions of the same application. Depending on task granularities, the information in the TiT may result in inaccuracies, as SG-SMPI depend on time measurements to estimate the FLOP count per task in the TiT. Instead of time measurements, the FLOP count per task can be measured using hardware counters, such as those exposed via the use of PAPI [BDG+00]. The FLOP count obtained with PAPI is used to represent the amount of work in each task in the simulation. The FLOP count per task is found to be a more accurate measurement to represent computational effort per task than time measurements and constant across different application executions [MEC+18a].

However, feeding the simulator the exact FLOP count per task might result in misrepresenting the dynamic behavior of native executions where task execution times vary among the different execution instances. A probability distribution is fitted to the measured tasks FLOP counts to address this misrepresentation. The simulator then draws samples from this distribution to represent the task FLOP counts during the simulation, as shown in the upper part of Figure 6.2.

Two different simulation approaches that exploit different SG interfaces for different purposes are presented herein and used in this work. The details of each approach are described next.

### 6.2.1 SimDag Simulation Approach

In SG-SD, the applications are represented as DAGs of tasks. Dependencies can be added between tasks to represent execution precedence. Tasks can be computation tasks or communication tasks. SG-SD is a general simulation interface that can be used to simulate parallel applications with multiple threads or multiple processes or both. It is not restricted to a particular programming language or model as long as an application is representable as a DAG of (dependent or independent) tasks. Each loop iteration is represented as a computation task [MEC18]. The amount of work in a computational task is equal to the FLOP counted by PAPI for the corresponding loop iteration. The FLOP count per iteration is read from a file to create computational tasks in the simulation that represent loop iterations. Alternatively, the FLOP count can be drawn from a distribution that represents the distribution of the FLOP count of the application tasks. Algorithm 6.1 shows how to create a SG-SD simulation and the main flow of execution to simulate the execution of an application with self-scheduling.

Whenever a PE is available, the scheduler calculates a chunk size and allocates it to this PE. A computation task and a communication task are created at each scheduling step to represent the scheduling overhead in calculating a chunk size and the communication with the requesting PE, respectively. The amount of work contained in the tasks denoting the computation scheduling overhead is acquired with PAPI to count the FLOP in the functions that calculate and assign chunks of work in the native code. The amount of communication in the tasks denoting the scheduling overhead is equal to eight bytes, which represents the communication of two integers (chunk size and the start index) typically required for nonadaptive DLS techniques. For adaptive DLS techniques, performance data are sent by the worker to the master per work request and a communication task with the corresponding amount of bytes is created in SG-SD to represent this communication.

**Algorithm 6.1 SG-SD simulation**

```
#include <simdag.h>
#include "DLS_scheduling.h"
/* Read input                                                    */
read_input(num_tasks, FLOP_file, platform_file, DLS_t, max_sim_t)
/* Create tasks that represent loop iterations                  */
Task_array = create_tasks(num_tasks, FLOP_file)
scheduled_tasks = 0
while (executed_tasks < num_tasks) && (get_sim_time() < max_sim_t) do
    idle_processes = get_idle_processes()
    for each idle_process in idle_processes) do
        /* Read and update finsihed tasks                       */
        executed_tasks += get_finished_tasks(idle_process)
        /* Send work request to master                          */
        send_work_request(idle_process, master)
        chunk = calculate_chunk(Task_array, num_tasks, scheduled_tasks, DLS_t)
        /* Assign work to worker                                */
        send_work(master, idle_process)
        scheduled_tasks += chunk
    /* Resume simulation untill a task is finished, i.e.,
       a process is idle                                        */
    simulate_execution(platform_file)
print("simulated time: " + get_sim_time())
print("finished tasks: " + executed_tasks)
```

### 6.2.2 SMPI+MSG Simulation Approach

Two interfaces of the SG toolkit are leveraged to simulate realistically the application performance with minimal effort [MEC+19]. This simulation approach is suitable for multiprocess parallel applications that use the message passing interface (MPI). Application processes need to be single-threaded. Algorithm 6.2 shows the changes needed in the native application code to transform it into the simulative application code using SMPI+MSG using the methodology illustrated in Figure 6.2. A single dynamic load balancing library, *DLS4LB* (described in Chapter 9), is used in both native and simulative codes. Lines in mint font color in Algorithm 6.2 show additions to simulate the application, lines in gray font color show the lines that need to be uncommented to revert to the native application code, and black lines denote unchanged code.

After the described code transformations in Algorithm 6.2, the code is compiled with the SG-SMPI compiler wrapper (*smpicc* or *smpif90*). The SG-SMPI interface is used to execute the native application code by using *smpirun* instead of *mpirun*. The computational tasks in the application are replaced with SG-MSG tasks to speed up the SG-SMPI simulation. The amount of work per SG-MSG task is either read from a file or drawn

**Algorithm 6.2 Native code transformation into SMPI+MSG simulative code**

```
#include <mpi.h>
#include "DLS4LB.h"
#include "msg.h" /* simulative only*/

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &P);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

/* Initialization */
...
/* results_data = malloc(N); native only*/
tasks = create_MSG_tasks(N); /* simulative only */
DLS_setup(MPI_COMM_WORLD, DLS_info);
DLS_startLoop (DLS_info, N, DLS_method);

t1 = MPI_Wtime();
while Not DLS_terminated do
    DLS_startChunk(DLS_info, start, size);
    /* Main application loop */
    /* Compute_tasks(start, size, data); native only */
    Execute_MSG_tasks(start, size); /* simulative only */
    DLS_endChunk(DLS_info);
DLS_endLoop(DLS_info);
t2 = MPI_Wtime();
print("Parallel execution time: %lf \n", t2 - t1);
/* Output or save results removed from simulation- native only */
...
MPI_Finalize();
```

from a probability distribution. Memory allocations of results and data in the native code are removed or commented in the simulation as they are not needed. This allows reducing the memory footprint of the simulation and the simulation of a large number of ranks on a single compute node. No modifications are needed for the *DLS4LB* in this approach. The scheduling overhead of different techniques is accounted for by the SG-SMPI, whereas the tasks execution time is accounted for in simulation by the SG-MSG. The proposed approach enables a fast and accurate simulation of the application with minimal modifications to the native application source code. Hundreds to thousands of MPI ranks can be simulated using a single core on a single compute node.

## 6.3 Representing Native Computing System Characteristics

Representing HPC systems in simulation involves representing different system components that contribute to applications' performance in simulation. The application and computing system representation cannot be seen as completely decoupled activities, i.e., representing a computing system should take into account the application characteristics as current simulators can not simulate all the complex characteristics of HPC systems precisely to create a general, application-independent system representation [MEC+18a]. For the simulation of the performance of computationally-intensive applications with different DLS, two main components of systems need to be represented:

1. The PEs, their number, and their computational speed;

2. The interconnection network between the PEs, the network bandwidth, the network latency, and its topology.

### 6.3.1 Processing Elements Representation

The PEs representation in simulation needs to reflect their native configuration in terms of the number of compute nodes and the number of PEs per node. Each core is represented as an SG host in the `platform file` to have full control over the behavior of every single core in the simulated system and how cores communicate. A host in SG `platform file` can be turned on or off to represent failures, and its availability can be changed between 0% to 100%. Therefore, perturbations in native execution (including failures) can be simulated at the core level by manipulating the availability or the state of the corresponding SG host. Hosts that represent the cores of the same node are connected with links (*loopback links*) with high bandwidth and low latency to represent the communication of cores of the same node through the memory. Similar to hosts, bandwidth and latency of links in SG can also be changed during simulation via changing their availability and state to represent irregular memory access times due to perturbations. To represent the fact that possible delays may occur if multiple cores are trying to access the memory at the same time, the sharing property of *loopback links* are set to SHARED (see Section 4.1)to represent possible memory contentions.

#### 6.3.1.1 Processing elements speed calibration

As a first step, nominal values for the PE computing speeds and the memory bandwidth and latency are added in the simulated HPC representation to obtain an initial representation. The second step is to fine-tune this initial representation to reflect the "real" HPC performance in executing a specific application. To this end, core speeds are estimated to obtain more accurate simulation results since applications do not execute at the theoretical peak performance. The core speed is calculated by measuring a loop execution time in a sequential run to avoid any parallelization or communication overhead. The sum of the total number of FLOP in all iterations in the loop is divided by the measured loop execution time to estimate the core processing speed. This core speed is used in the simulated HPC representation to reflect the native core speed in processing the application tasks [MEC+18a]. The estimation of PE core speed is performed per PE type in heterogeneous HPC systems, to estimate the speed of each different CPU type in the system.

A memory benchmark, such as Stream benchmark [McC95], can be used to estimate the memory bandwidth and latency. These values are inserted in the *loop-back links* bandwidth and latency.

### 6.3.2 Network Representation

Another set of links is used to connect the hosts to represent the network topology of an HPC system. The properties of these links (bandwidth and latency) represent the properties of the interconnect fabrics of the simulated HPC system, such as InfiniBand or Omni-Path fabrics. To reflect the fact that network communications are nonblocking in a native HPC system, the `FATPIPE` is used to configure SG that the communications on these links are nonblocking and is not shared, i.e., each host has all network bandwidth and the shortest latency available at all times even in the case of all hosts are communicating at the same time.

#### 6.3.2.1 Network calibration

Similarly, a simple network benchmarking, such as a ping-pong test could be used to estimate the real network links communication bandwidth and latency, and then insert these values in the simulation. SG also provides a tool for the fine-tuning of the network properties in a `platform file`. The SG-based calibration procedure [Sim14] can be used to calibrate the representations of network in `platform file`s. For example, the SG-based calibration procedure generates certain calibration parameters to adjust the network bandwidth and latency according to different message sizes as shown

in Figure 6.4. This accounts for the use of different algorithms by the MPI to send small and large messages.

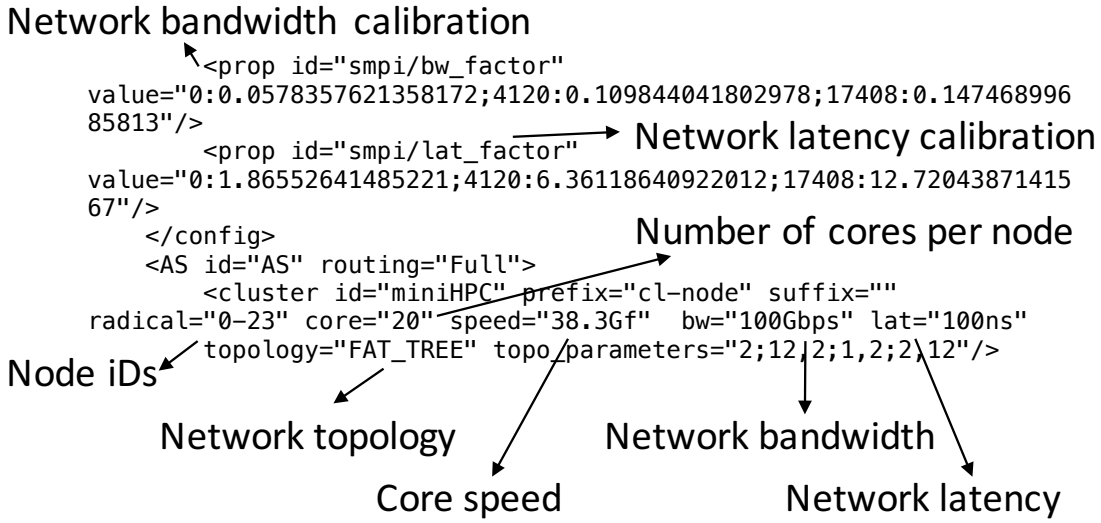



**Figure 6.4** **A sample calibrated SG `platform file`.** The file is annotated to show the added calibration parameters by the produced SG calibration procedure.

### 6.3.3 System Variability

Quantifying system variability is challenging due to the variety of factors that cause the variability, e.g., system failures, operating system kernel interrupts, memory, and network contentions [SK05]. An approach for quantifying the variability in the system from application performance was proposed [SMW18]. The application is divided into computation segments, where the performance is analyzed between communication calls. The execution times of such segments are classified to identify intrinsic and extrinsic application performance variations. Therefore, the amount of system noise can be measured based on the measured extrinsic performance variations in application segments.

Alternatively, the effect of the system variability on application performance can be examined by exploiting a backlog of application execution times [BGG18]. Two factors, *maximum perturbation level* $PL_{max}$ and *minimum perturbation level* $PL_{min}$, are used to determine the upper and the lower bounds of a uniform distribution $U$ used to estimate the *perturbation level* $PL$ induced by the system. These factors are calculated as in Equations 6.1 and 6.2, where $E_i$ denotes the application execution time at the $i^{th}$ execution instance and $E\prime$ is the average application execution time of $n$ execution instances.

$$PL_{max} = \max_i \left( \frac{|E_i - E\prime|}{E\prime} \right) \tag{6.1}$$

$$PL_{min} = \min_i \left( \frac{|E_i - E\prime|}{E\prime} \right) \tag{6.2}$$

The estimated $PL$ is calculated as in Equation 6.3 and can be used to perturb PE availabilities during simulation, i.e., the performance variability observed during a native execution of a parallel application is injected in the simulation [MEC+19].

$$PL = U\left[PL_{min}, PL_{max}\right] \tag{6.3}$$

### 6.3.4 Verification of the Computing System Representation

The representation of the computing system can be verified in a separation of the application representation by using the SG-SMPI interface. The SG-SMPI interface simulates the execution of native MPI codes on a simulated computing `platform file`. Both the native and simulative executions using SG-SMPI share the application's native code. The difference between the native execution and the simulative SG-SMPI-based execution is the computing system representation component. The representation of the computing system can be verified by comparing the native and SG-SMPI simulative performance results (see Figure 6.5).

## 6.4 Visualizing Simulative Executions

Visualizing execution traces plays a crucial role in performance analysis. Therefore, simulative execution trace visualization is as important as native execution trace visualization. To provide similar native and simulative traces visualizations, SG can be configured to print out the start and finish times of each task, and on which host (PE) a task was executed in the form of text-based execution trace. A tool [Yes16] was created to convert the collected text-based traces to binary traces in the open trace format (OTF2) [EWG+11]. Using OTF2 traces with the Vampir [KBD+08] trace visualizer, we are able to visualize simulative execution traces similar to native execution traces.

The comparison of native and simulative execution traces can be used to verify that simulators not only produce execution times similar to native execution but also reproduce the same load imbalance in the native execution. This is crucial for the study of scheduling and load balancing using realistic simulations. For example, this tool was used to generate a simulative execution trace of Adjoint convolution with decreasing

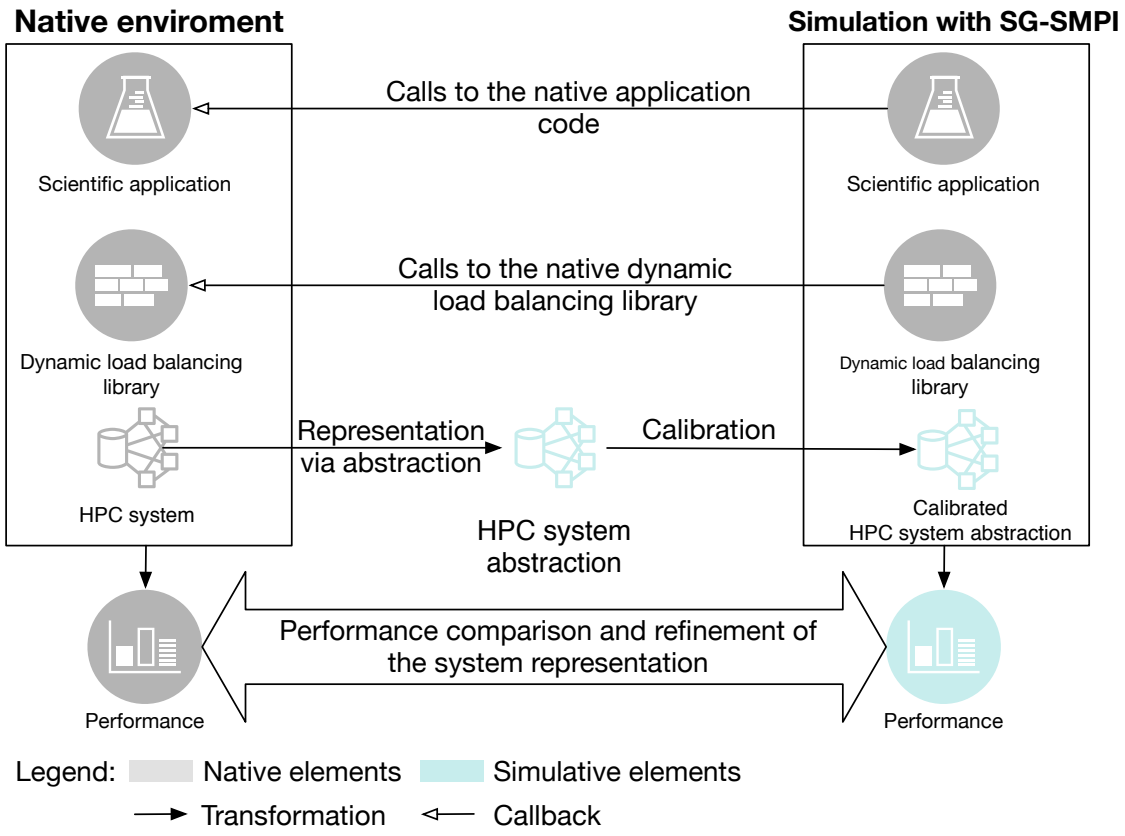
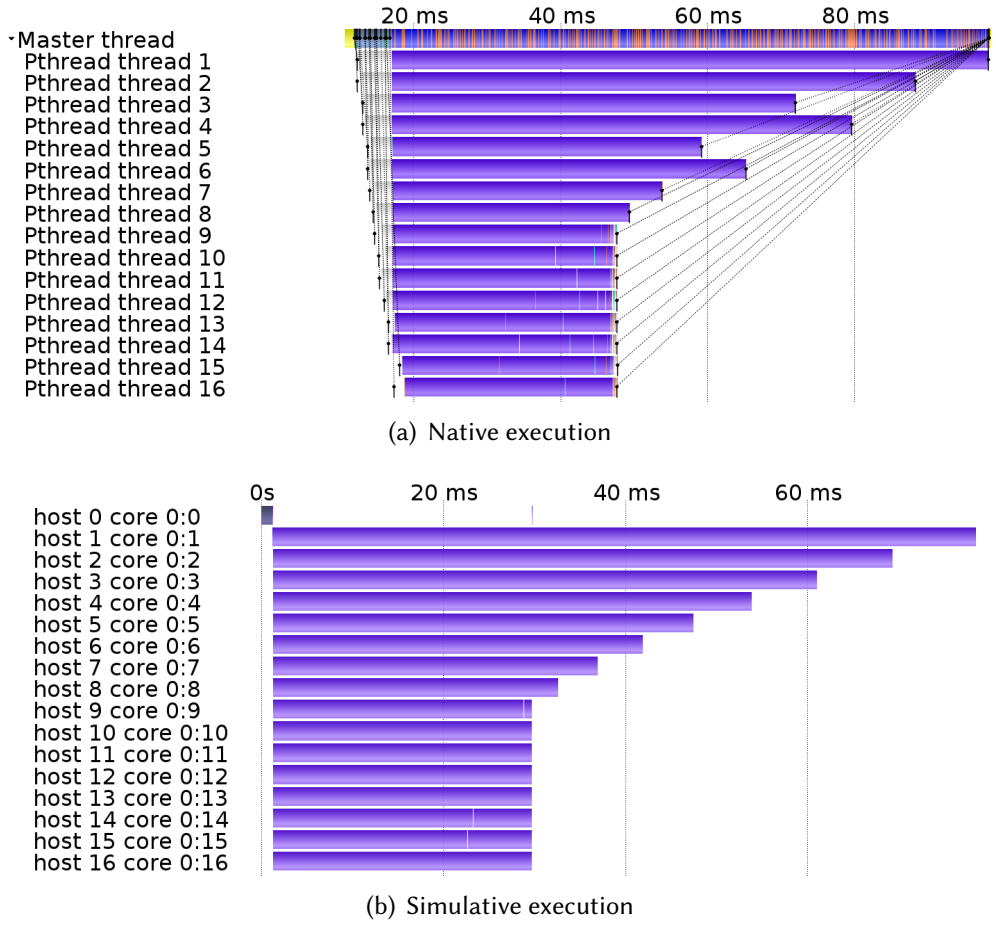



**Figure 6.5 Computing system representation verification using SG-SMPI.** SG-SMPI leveraged as a midpoint between fully simulative execution (Figure 6.2)) and the native execution to verify one component of the simulation, i.e., computing system representation [MEC17a].

task sizes (AD-d) (see Section 7.2) with 16 threads on miniHPC (see Section 8.2) and is compared with the native execution trace in Figure 6.6. The native trace is generated by instrumenting the code for tracing with Score-P [KRM+12] and tracing the application execution. The output trace is then visualized using Vampir. The comparison shows that the simulator successfully reproduces, both, the load imbalance in the native execution as well as the native execution time.




(a) Native execution

(b) Simulative execution

**Figure 6.6** **Comparison of native and simulative execution traces.** Traces obtained from execution (top) (with execution time of 0.082 s) and its corresponding simulation (bottom) (with simulated execution time of 0.079 s) of the GSS - AD-d using 16 worker threads. The comparison shows that the simulator accurately reproduces the load imbalance of the native execution [MEC17b].

# 7

# Verification of Selected Dynamic Loop Scheduling via Reproduction of Experiments in Simulation

A number of DLS techniques have been proposed between the late 1980s and early 2000s and efficiently used in scientific applications [BV02; BFH95; BWA16; CB07; EMC17a] (see Figure 1.7). In most cases, the computing systems on which they have been tested and validated are no longer available. Therefore, it is essential to ensure that the DLS techniques employed in scientific applications today adhere to their original design goals and specifications. Verification ensures the minimization of the sources of uncertainty in the implementation of DLS techniques to avoid unnecessary influences on the performance of scientific applications.

The goal of this chapter is to verify of the present implementation of DLS techniques. To achieve this goal, the performance of a selection of scheduling experiments from the 1992 original work that introduced *factoring* (FAC) [FHSF92] is reproduced via simulative experimentation (see Figure 7.1). Simulation methods described in Chapter 6 are employed herein for the reproduction of scheduling experiments on the HPC system in 1992.

Reproducibility is a key aspect of the scientific method [ACM16]. The reproduction of scientific experiments contributes to the validation of those experiments and to establish that the conclusions drawn from these experiments are of scientific relevance [HT13]. *Reproduction* [ACM16] is defined as revisiting a particular scientific problem, namely, the performance of DLS techniques, without the original artifacts or the possibility to execute the artifacts on the original computing system [PBG+85]. Reproduction is employed herein as a means to attain and increase the trust in native and simulative implementations of DLS.

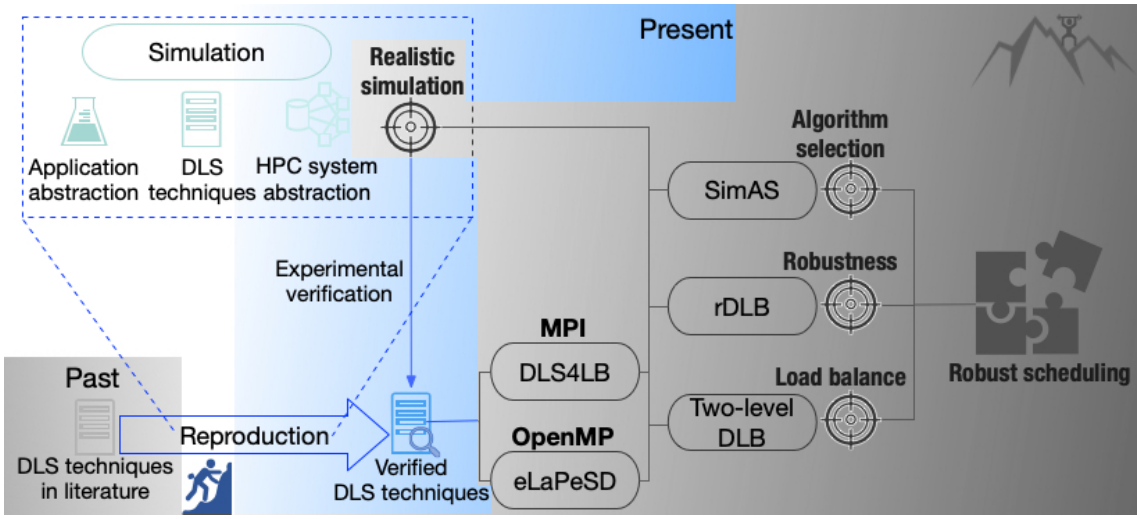



**Figure 7.1 Illustration of the focus in this chapter (in colors) as part of the overall approach (in grayscale).** Simulation is used for the reproduction of DLS experiments of the past. DLS implementation is experimentally verified by comparing original results of the past with reproduced simulative results in the present (see Section 7.1).

We answer *"How to ensure that the DLS techniques employed in scientific applications today adhere to their original design goals and specifications?"* We explain how original experiments were simulated and discuss and compare the original and reproduced performance results. Verified DLS implementations are used in the next chapters in native and simulative experiments to examine the load balancing of scientific applications under various execution scenarios.

## 7.1 Verification via Reproduction Approach

We devised a reproduction and verification approach (shown in Figure 7.2) for the realistic simulation, reproduction, and verification of the performance of scientific applications with DLS techniques on HPC systems. It consists of three steps. First, we reproduce selected experiments [FHSF92] from the past using simulation to verify the present implementation of DLS techniques [MEC18]. Second, the same simulation is also used to reproduce native experiments from the present and are compared to simulative results to compare and verify the predictions of the simulator. Third, various simulation approaches are compared for their prediction of application performance relative to the native performance (ground truth) to stand on how realistic are performance simulations and how their predictions are affected by various factors.

This chapter implements the first step in the reproduction and verification approach described in Figure 7.2 by comparing present simulative experiments to past native ex-

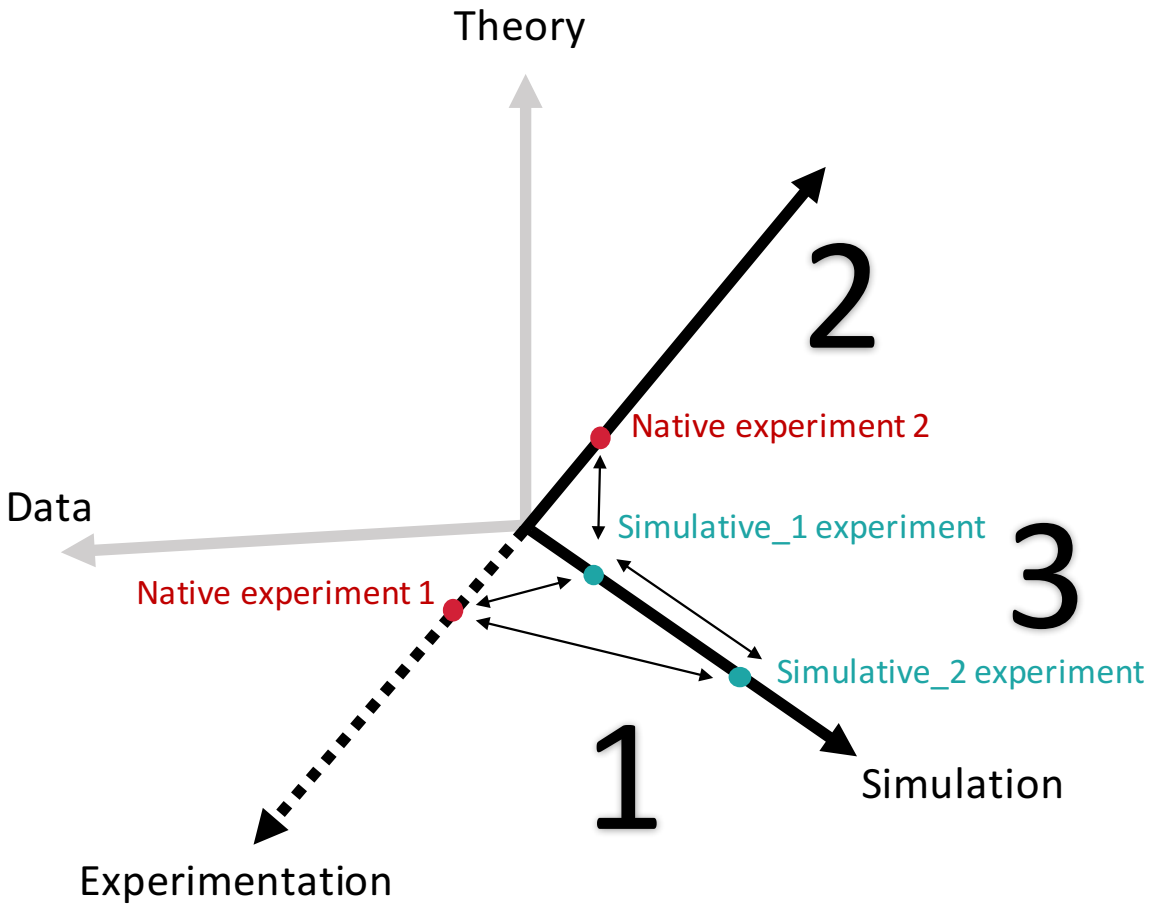



**Figure 7.2 Illustration of the reproduction and verification approach.** Performance comparisons over the pillars of science are employed in this work for the verification of the DLS techniques: (1) Native experiments from the past original work from the literature are reproduced in present simulations to verify DLS techniques implementation. (2) Simulative and native results from experiments in the present are compared to verify the fidelity of realistic performance simulations. (3) Different simulation approaches are compared to achieve close agreement in terms of simulation of application performance to that of the native performance.

periments to verify the present implementation of DLS techniques. The scientific challenge is the reproduction of the performance of the past experiments with incomplete information, such as the computing system characteristics and the implementation details. Steps two and three of the reproduction and verification approach are implemented in Chapter 8.

## 7.2 Selected Experiments for Reproduction

A selection of scheduling experiments from the 1992 original work that introduced FAC, one of the efficient DLS techniques proposed for shared-memory systems, are considered herein for reproduction via simulative experimentation. Therein, the performance of the IBM Research Parallel Processor Prototype (hereafter, the RP3) system [PBG+85] was compared for the execution of three computational kernels: matrix multiplication (MM), adjoint convolution (AC), and Gauss-Jordan elimination (GJ). Four loop scheduling techniques: STATIC, SS, GSS, and FAC, were used to execute these three kernels. In this chapter, the scheduling behavior of the first *two* computational kernels using the above *four* scheduling techniques is reproduced to confirm that the implementations of the DLS techniques adhere to their original goals and specifications. Several factoring-based DLS techniques have been proposed since 1992 to strike the best balance between increased load balance and decreased scheduling overhead, such as weighted factoring [FHSU+96], adaptive weighted factoring [BVD03], and adaptive factoring [BL00]. Confirming the adherence of the STATIC, SS, GSS, and FAC implementations to their original design goals lays the foundation for confirming the implementation of further DLS techniques, based upon the factoring.

### 7.2.1 Selected Applications

The matrix multiplication (MM) and adjoint convolution with decreasing task sizes (AC-d) kernels are selected for reproduction and prediction, with matrix sizes of $300 \times 300$ and $75 \times 75$, respectively. The computational kernels are described in Algorithms 7.1 and 7.2. Changing the loop direction in Algorithm 7.2 at Line 1 will result in tasks with increasing sizes rather than decreasing sizes, i.e., AD-i. AD-d is considered for reproduction as it more challenging (load balancing wise), as GSS and FAC assign chunks of decreasing sizes as well. Large chunk sizes of large task sizes in AD-d assigned by GSS and FAC could lead to imbalanced execution.

The two kernels represent two different task granularities: equal task sizes and decreasing task sizes. Each iteration of a kernel's *for loop* was considered a task to be scheduled. For instance, Lines 5-6 in Algorithm 7.1 represent one task of the matrix multiplication kernel, and Lines 3-4 in Algorithm 7.2 represents one task of the adjoint convolution kernel with decreasing task sizes.

**Algorithm 7.1 Parallel matrix multiplication (MM)**

**Input**: Matrices $A$ and $B$ each of size $n \times n$
**Output**: Matrix $C$ of size $n \times n$
**Data**: $A$, $B$, $n$
**Result**: $C \leftarrow A \times B$

```
1 for k = 1 : n × n do in parallel
2     i ← k/n
3     j ← k − n × (k − 1)/n
4     C[i, j] ← 0
5     for l = 1 : n do
6         C[i, j] ← C[i, j] + A[i, l] × B[l, j]
```

**Algorithm 7.2 Parallel adjoint convolution (AC-d)**

**Input**: Two matrices $A$ and $B$ each of size $n \times n$
**Output**: Matrix $C$ of size $n \times n$, where $C$ the adjoint convolution of $A$ and $B$
**Data**: $A$, $B$, $C$, $n$, $const$

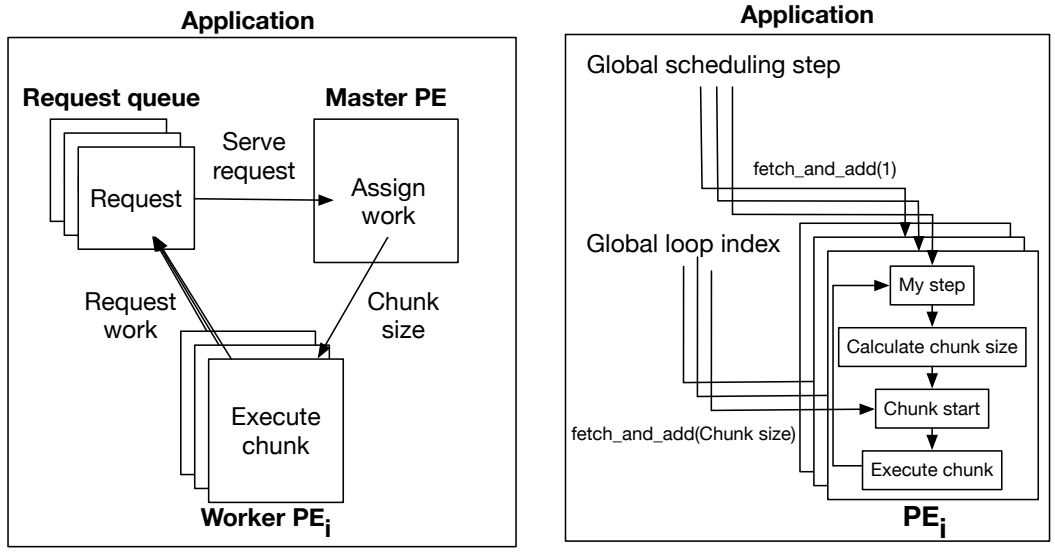
```
1 for k = 1 : n × n do in parallel
2     C[k] ← 0
3     for l = k : n × n do
4         C[k] ← C[k] + const × A[l] × B[l − k]
```

### 7.2.2 Centralized Versus Decentralized Coordination DLS Implementation

Investigating the implementation of DLS techniques, we found two approaches that can be employed to implement process coordination in the DLS techniques natively or in simulation: (1) Centralized coordination, using a master-worker execution model, and (2) Decentralized coordination, wherein each "worker PE" calculates and executes a chunk of work whenever it becomes available. Figure 7.3 shows both implementation approaches. Table 7.1 summarizes the advantages and disadvantages of both implementations.

Centralized coordination DLS using the master-worker model is simple to implement as only one PE accesses and updates the central self-scheduling work queue. However, the master might become a performance bottleneck with a frequent and large number of requests from workers. Also, if the master fails, the whole application fails. Decentralized coordination overcomes the above limitations of the centralized coordination at the cost of more complex implementation and the need for data sharing (within nodes and across nodes for threads and processes, respectively) and synchronization mechanisms.

The flow of the master and worker programs in the centralized coordination DLS are described in Algorithm 7.3 and Algorithm 7.4, respectively. The master implements




(a) Centralized coordination approach (b) Decentralized coordination approach

**Figure 7.3 Illustration of the centralized and decentralized coordination approaches of DLS techniques.** Centralized coordination approach depends on a centralized master that handle work requests from (worker) PEs. In the decentralized coordination approach, each PE allocates to itself a chunk of work from a shared global pool of work. Synchronization via locks or atomics is needed to access the work pool data structure from different parallel PEs simultaneously [MEC18].

**Table 7.1 Advantages and disadvantages of centralized and decentralized coordination implementations of DLS techniques**

| Coordination approach | Centralized | Decentralized |
|---|---|---|
| Implementation | Master-worker model<br>Master handles worker requests | Each PE calculates and allocates<br>chunks for itself from shared work pool |
| Advantages | Simple to implement<br>No data race issues | Improved scalability<br>Reduced synchronization overhead<br>Robust, no single point (the master) of failure |
| Disadvantages | Master is a performance bottleneck<br>Master is a single point of failure<br>Poor scalability | Needs complex synchronization mechanisms |

the scheduling algorithm and, hence, it is responsible for assigning chunks of tasks to worker PEs. Each worker implements the main computation kernel of the application. Whenever free, worker PEs push work requests to the master and wait to be assigned work. After finishing the assigned work, PEs push work requests and wait to be assigned work again, until they receive a terminate signal from the master, indicating that all work is finished.

The program flow of the PEs in the decentralized coordination approach is described

**Algorithm 7.3 Centralized coordination DLS – Master**

**Input**: *numTasks*, *DLSmethod*, *numWorkers* ← number of worker PEs
/* *data* is a structure that contains two elements: *start* to specify the start of the chunk of tasks, and *chunkSize* to specify the size of the chunk */
/* *requests* is a work request queue shared between master and worker PEs */
**Data**: *data*[*numWorkers*], *requests*, *scheduledTasks*, *free*[*numWorkers*]
1 **while** *scheduledTasks* < *numTasks* **do**
2  **foreach** *i* ∈ *requests* **do**
3  *chunkSize* ← calculate_chunk(*numPE*, *numTasks*, *scheduledTasks*)
 /* assign chunk to worker with *PEID* = *i* */
4  *data*[*i*].*chunkSize* ← chunkSize
5  *data*[*i*].*start* ← *scheduledTasks*
 /* increment *scheduledTasks* */
6  *scheduledTasks* ← *scheduledTasks* + *data*[*i*].*chunkSize*
7  *free*[*i*] ← *False*
8 wait for all worker PEs to complete execution
9 send terminate signal to all worker PEs
10 terminate the program

**Algorithm 7.4 Centralized coordination DLS – Worker**

/* *data*, *free*, and *requests* are shared between master and worker PEs */
1 i ← PEID
**while** *true* **do**
 /* make an new request */
2  *free*[*i*] ← *True*
3  *Push*(*i*) → *requests*
4  wait until *free*[*i*] = *False*
5  **if** *data[i].chunkSize = -1* **then**
6  break /* terminate */
7  **else**
8  **foreach** *task in the assigned chunk* **do**
9  execute_computation(*start*, *chunkSize*)

in Algorithm 7.5. Each PE obtains work using the `obtain_work` function described in Algorithm 7.6. The program holds two global variables that represent the current state of the program: `schedulingStep` and `currentIndex`. The `currentIndex` represents the loop index of the outer loop that is parallelized of the computational kernels and indicates the program progress. The `obtain_work` function updates these two variables after each work assignment to advance the program state. Updates

to these global variables (Lines 1 and 3 in Algorithm 7.6) are performed using *atomic* operations to avoid data races between parallel PEs. The usage of atomic operations avoids performance degradations that may be caused by using *locks*. Atomic operations can ensure uninterrupted safe operations (i.e., read-modify-write operations) on global variables from multiple parallel PEs.

**Algorithm 7.5 Decentralized coordination DLS — PE execution**

```
Input: PEID
Output: void
Global data: method, schedulingStep, currentIndex
Local data: start, chunkSize
1 while True do
2     if obtain_work(start, chunkSize) then
3         execute_computation(start, chunkSize)
4     else
5         break
```

**Algorithm 7.6 Decentralized coordination DLS — obtain work**

```
Input: method, schedulingStep, currentIndex
Output: start, chunkSize
Global data: method, schedulingStep, currentIndex, numTasks
Local data: myStep
1 myStep ← fetch_and_add(schedulingStep, 1)
2 chunkSize ← calculate_chunk(numPE, numTasks, myStep)
3 start ← fetch_and_add(currentIndex, chunkSize)
4 if start < numTasks then
5     if start + chunkSize >= numTasks then
6         chunkSize ← numTasks − start
7     return True
8 else
9     return False
```

DLS techniques were implemented originally using decentralized process coordination. For completeness, both implementations, centralized and decentralized coordination approach, are discussed and compared herein.

### 7.2.3 Computing System of the Reproduced Experiments

The scheduling experiments were performed on the RP3 system [PBG+85] in the original work [FHSF92]. The RP3 configuration by IBM was designed to accommodate up to 512 processors to reach the performance of 800 MFLOP/s and 13 GB/s inter-processor communication speed. Therefore, a single processor speed is 1.562 MFLOP/s. The main

building unit of the RP3 was the processor-memory element (PME). Figure 7.4 shows an overview of the architecture of the RP3 system.

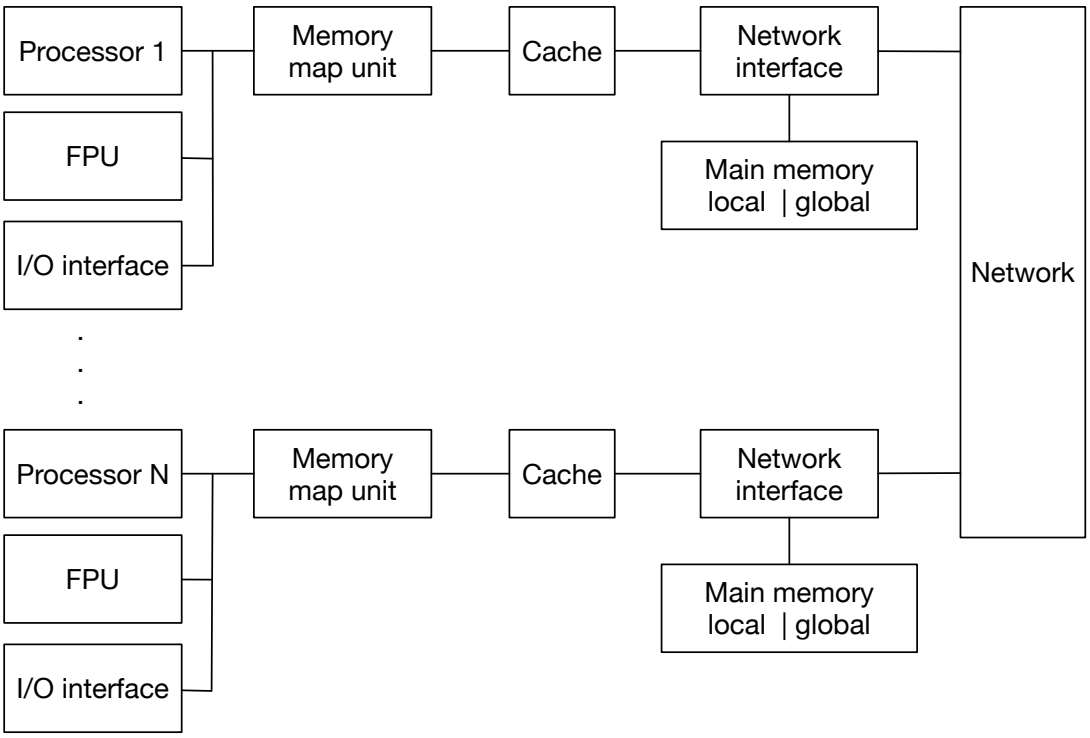



**Figure 7.4 An overview of the RP3 architecture.**

Each PME contained a 32 bit microprocessor, 2-4 MB of memory, 32 KB of cache, a floating-point unit that supporting 32 bit and 64 bit operations, an I/O interface, and a memory address translation unit. The shared memory space was marked not to be cached to avoid cache coherency issues. PMEs were connected using the Omega network, with network link bandwidth of 50 Mbit/s, and latency of 2 $\mu$s [PBG+85]. Each memory reference was sent to the memory address translation unit to decide if the reference is in its local memory, or if the request needs to be sent over the network to retrieve the data from other PMEs.

The RP3 system was operated by a version of the Berkeley Software Distribution (BSD) UNIX operating system, with unique extensions made to support the parallel architecture of the RP3 system and its non-coherent caches [BCR91]. The four scheduling methods were implemented as a part of the IBM RP3 runtime system for the IBM restructuring Fortran compiler PTRAN [ABC+88].

### 7.2.4 Simulation of the Selected Experiments

The selected experiments details are summarized in Table 7.2. To confirm the implementation of the four scheduling techniques, the selected scheduling experiments on the

RP3 system [FHSF92] are reproduced and compared with those obtained using SG-SD. SG-SD simulation approach (described in Section 6.2.1) is selected as the original experiments were performed on a shared-memory systems using threads. Therefore, the SG-SMPI simulation approach would not be suitable for the reproduction of the selected experiments.

**Table 7.2 Selected scheduling experiments**

| Computational kernel | Matrix size | Scheduling method | Coordination | Number of processors |
|---|---|---|---|---|
| Matrix multiplication (MM) | $300 \times 300$ | STATIC, SS GSS, FAC | Centralized Decentralized | 4, 8, 16, 24, 32, 40, 48, 56 |
| Adjoint convolution with decreasing task sizes (AC-d) | $75 \times 75$ | | | |

Every iteration of an MM and AC-d computational kernel's outer loop is modeled as an SG-SD *sequential computation task*. The amount of work contained in each computational task is specified in FLOP count per loop iteration in the simulator. For both MM and AC-d, the FLOP count in each iteration is inferred from their pseudocodes in Algorithms 7.1 and 7.2. This number is used in the simulator as the amount of work in each sequential computation task.

An SG-SD *sequential computation* task is created to represent the scheduling overhead of each DLS technique. This task is scheduled on the available PE in each simulated scheduling round. The amount of work performed by each of these scheduling tasks varies and depends on the selected scheduling technique. The values for the amount of work performed by each of these scheduling overhead tasks are obtained empirically, to match the simulation results to the results in the original publication [FHSF92]. Specifically, they are found to be 75, 400, 750, and 750 FLOP for STATIC, SS, GSS, and FAC techniques, respectively. An SG-SD *end-to-end communication task* is also created in each scheduling round to simulate the time taken to copy the assigned chunk of tasks from the central work pool to the available PE that needs work. It is assumed that, initially, PE 0 stores all the data, and other PEs transfer one column of the matrix from PE 0 for every task they obtain. This data strategy is referred to as *pool of tasks and data*.

The amounts of computation (FLOP) and communication (Byte) in each loop iteration for the two selected computational kernels are presented in Table 7.3. Examination of the FLOP count in the MM in Algorithm 7.1, reveals that each iteration of the outer loop comprises 5 FLOP before the innermost loop and 2 FLOP that are repeated on all the elements of a row of the input matrix. Similarly, for AD-d, the innermost loop only contains 3 FLOP. The innermost loop is repeated a number of times that is equal to the difference between the matrix size and the iteration counter. Two factors, $g_1$ and $g_2$, are

used to capture the unknown effects in the execution of the computational kernels on the RP3 system. These factors cover all software- and hardware-related aspects that may influence program execution on RP3, e.g., memory system and operating system interference. These factors are presented in Table 7.3, are unitless and are experimentally determined to be 35 and 60, respectively.

To provide the SG simulation engine with the specifications of the simulated system, it requires a `platform file` to specify the major components of the hardware and their properties. Each processor in the RP3 system is represented as a host in the SG `platform file` used in the reproduction experiments. All hosts (processors) are interconnected by creating a communication link between every host and all others. Details of the RP3 system are extracted from the work that introduced the RP3 system [PBG+85]. The values used to describe the RP3 system in the developed simulator are presented in Table 7.4.

All simulations are performed using SG-SD 3.16 on a manycore compute node with an Intel KNL processor (7210) running at 1.3 GHz, using CentOS operating system, version 7.2.1511. The GNU C compiler, version 6.3.0, is used for the compilation of the simulator with `-g -Wall` as compilation flags.

## 7.3 Verification via Reproduction Results

The simulative performance for executing MM and AC-d using centralized (SG-SD-C) [MEC17b] and decentralized (SG-SD-D) [MEC18] coordination approach with SG-SD compared against the original native performance results [FHSF92] is illustrated in Figure 7.5. The selection of parallel cost as a performance metric (over the parallel execution time) is since the parallel cost was used in the original publication [FHSF92]. The parallel cost reflects the sum of the time that PEs spend solving the problem [KGG+94].

The results show that the simulation performance is close to the native performance in the original publication. The percent error ($\%E$) between the simulative execution time in this work ($T_{sim}$) and the original native execution time ($T^{o}_{nat}$) [FHSF92] is calculated

**Table 7.3 Computational kernels parameters for their simulation on the RP3 system.**

| Computational kernel | Task size (FLOP) | Communication size (Byte) |
|---|---|---|
| MM | $g_1 \times (5 + 2 \times \text{rowLength})$ | $\text{chunkSize} \times \text{rowLength}$ |
| AC-d | $g_2 \times 3 \times (\text{matrixSize} - \text{iterationID})$ | $\text{chunkSize} \times \text{rowLength}$ |

**Table 7.4 SG platform description for simulating the performance of the two computational kernels on the RP3 system**

| Parameter | Value |
|---|---|
| Processor speed | 1.562 MFLOP/s |
| Interconnection link bandwidth | 50 Mbit/s |
| Interconnection link latency | 2 us |

as:

$$\%E = \left(1 - \frac{T_{sim}}{T^{o}_{nat}}\right) \times 100. \tag{7.1}$$

A positive percent error $\%E$ indicates that the simulator underestimates the original execution time, while a negative $\%E$ signifies overestimation. For the results of the decentralized process coordination, the minimum absolute $\%E$ between SG-SD-D and the native execution is 0.073%, for GSS — AC-d and 56 threads, as can be observed from Figure 7.5(g). The maximum absolute $\%E$ is 45.89% in the case of SS — MM and 4 threads, as can be observed from Figure 7.5(b). The average of the absolute $\%E$ is 10.89% in all the scheduling experiments on the RP3 system and the SG-SD-D simulation results [MEC18].

For the results of the centralized process coordination [MEC17b], the minimum and the maximum absolute $\%E$ are 0.49%, and 30.94%, respectively, in the case of GSS — AC-d and 24 threads (see Figure 7.5(g)) and SS — AC-d and 56 threads (see Figure 7.5(f)). The average of the absolute $\%E$ is 7.44% between the simulative results [MEC17b] (SG-SD-C) and the native execution results [FHSF92]. The simulative results follow a similar trend to the original native experiments, which is of high relevance for the comparison of different scheduling techniques.

### 7.3.1 Discussion

The reproduction of the behavior of the two computational kernels, MM and AC-d, has been used to confirm the adherence of the present implementation of four scheduling techniques to the original specification [FHSF92]. **The results *confirm* that the implementation of the considered DLS techniques in SG-SD adheres to their implementation used in the original publication [FHSF92]**.

The achieved trust in the implementation of DLS techniques for shared-memory systems has also been transferred to their implementation for distributed-memory systems [MEC+18a]. Confirming the adherence of the DLS implementation to their original design goals minimizes the sources of uncertainty in their implementation and helps to avoid unnecessary influences on the performance of scientific applications. For instance, a DLS that has been implemented to use shared memory locks intensively will cause un-

necessary scheduling overhead, adversely influencing performance. It is, therefore, of high value and significance to compare reproduced experiment results with original experiments results in the view of its experimental validation.

This concludes the first step of the reproduction and verification approach described in Figure 7.2 and confirms the present implementation of STATIC, SS, GSS, and FAC, which are the basis of more advanced DLS techniques summarized in Table 2.2. The verified DLS implementations are used in the next chapters in native and simulative experiments to examine the load balancing of scientific applications under various execution scenarios.

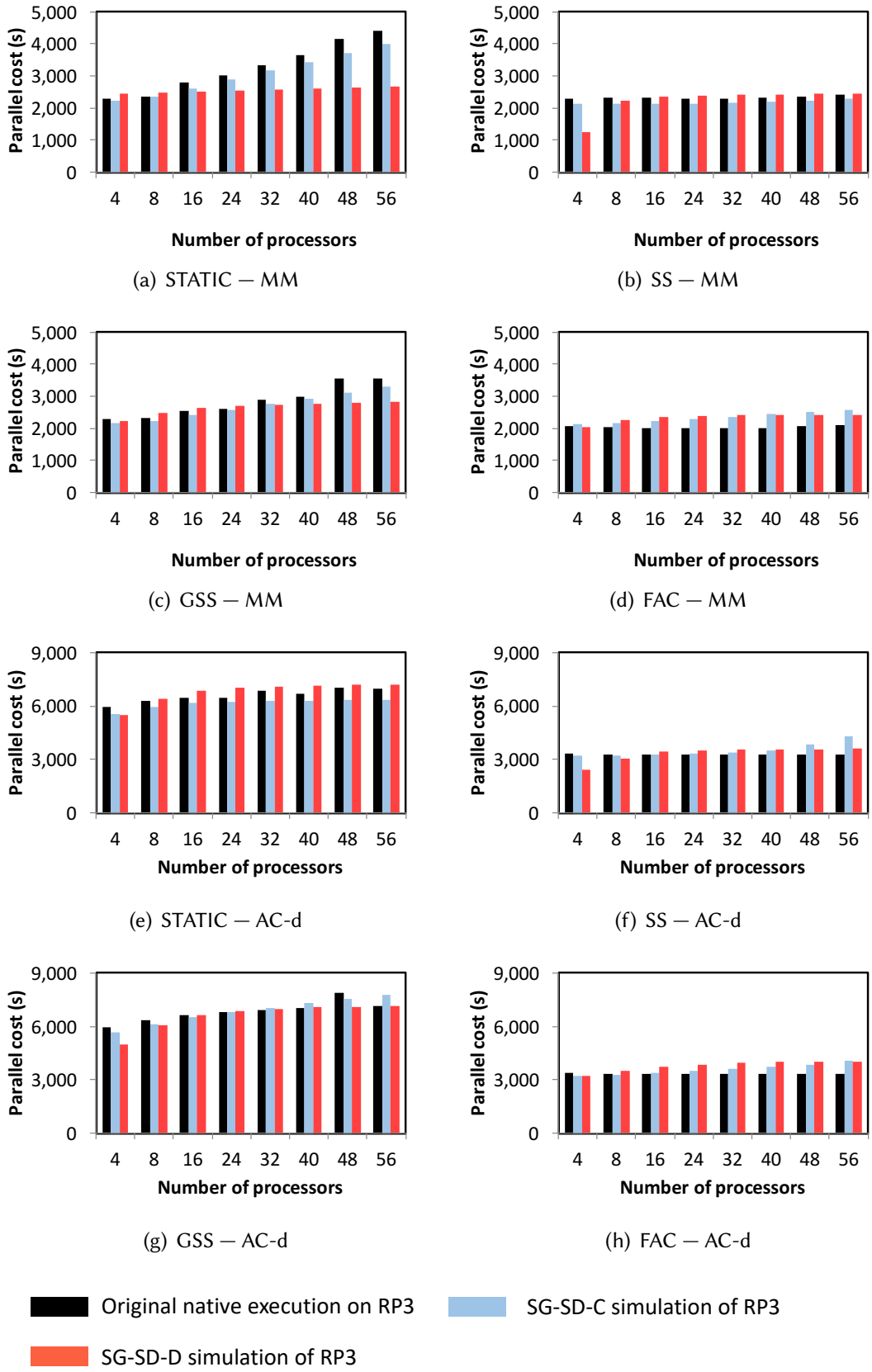



**Figure 7.5 Performance results for the selected DLS experiments on the RP3 system.** Simulation results using a decentralized process coordination (SG-SD-D) obtained with SG-SD (red bars) compared with the simulation results for the selected DLS experiments on the RP3 system using a centralized process coordination (SG-SD-C) [MEC17b] (blue bars) and the original publication [FHSF92] results (black bars). Parallel cost = parallel program execution time × number of PEs [MEC18].

# 8

# Verification of Simulation of Applications Performance on Modern Architectures

Simulation is the third pillar of science after theory and experimentation. However, conducting realistic and trustworthy simulations of application performance under different configurations is challenging. Several approaches to represent the application tasks (loop iterations) and computing system characteristics are presented in Chapter 6. In this chapter, we perform several scheduling experiments on native modern computing systems. We employ the simulation methods presented in Chapter 6 to simulate the performance of the native applications in the scheduling experiments and compare the native and simulative performance results. We compare native and simulative performance results to evaluate how realistic are performance simulations of scientific applications with DLS on modern HPC systems (see Figure 8.1).

This implements the second and third steps in the reproduction and verification approach described in Figure 7.2. The comparison of native and simulative performance results evaluates how close are they, *quantitatively* and *qualitatively*. Also, different simulations of the same experiment are performed. The performance results of different simulations are also compared to understand the effect of different simulation parameters and their effect on the simulative performance. Therefore, experiments presented and discussed in this chapter show and clarify how close is the simulation of applications' performance to native performance and the effect of each simulation choice presented in Chapter 6 on the close agreement between native and simulative performance.

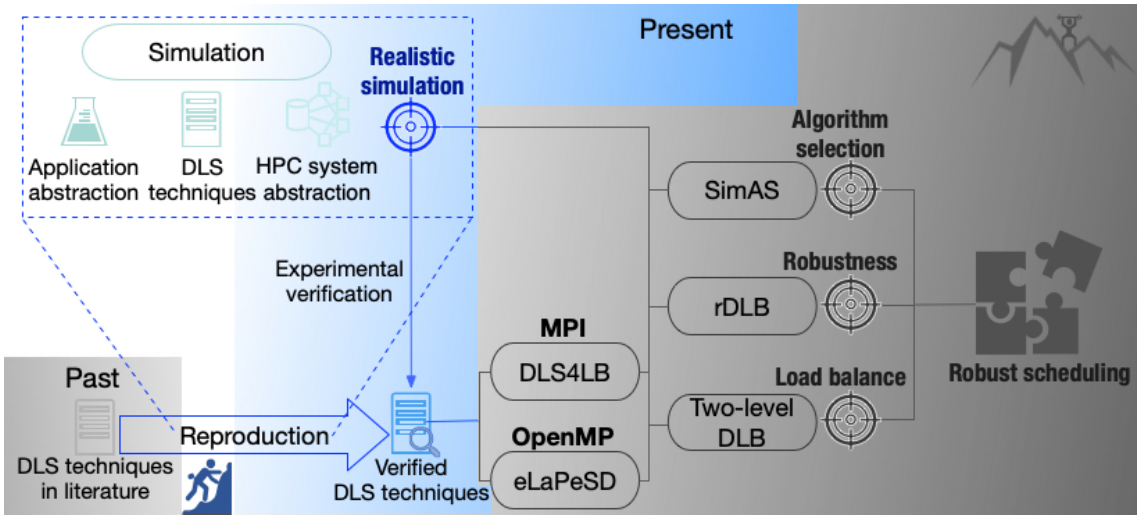



**Figure 8.1 Illustration of the focus and the progress towards robust scheduling (in colors) as part of the overall approach (in grayscale).** Verified DLS implementations are leveraged in native and simulative experiments on modern HPC systems. Performance of native and simulative experiments are compared to estimate how realistic is performance simulation.

## 8.1 Scientific Applications for Native and Simulative Experiments

Two applications are considered in the experiments in this chapter. The two applications represent two different cases of tasks. The first application (PSIA) incurs low variability between its parallel tasks, and the second one (Mandelbrot) incurs high variability among its parallel tasks.

### 8.1.1 PSIA

The first application is an application from the computer vision domain, namely, the parallel spin-image algorithm (PSIA) [EFM+16]. The SIA is a computationally-intensive application. The core computation of the SIA is the generation of the 2D spin-images. Figure 8.2 shows the process of generation of a spin-image for a 3D object.

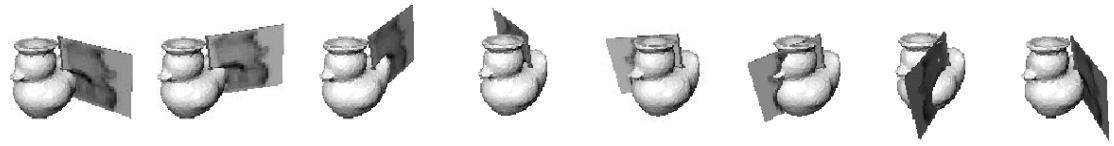


**Figure 8.2 Illustration of the spin-image calculation for a 3D object.** A flat sheet is rotated around each point of the 3D object to describe the object from this point view (from literature [Joh97]).

The PSIA exploits the fact that spin-images generations are independent of each other. The size of a single spin-image is small (200 bytes) and fits in the lower level (L1) cache. Therefore, the memory subsystem has an insignificant impact on applications' performance. Algorithm 8.1 list the pseudocode of the PSIA [MEC+18b]. According to Algorithm 8.1, Lines 9 and 12, the amount of computations to generate spin-images is data-dependent and not identical over all the spin-images generated from the same object. This variation introduces an algorithmic source of load imbalance among the parallel processes generating the spin-images. The number of spin-images generated by each PE is governed by the `start` and `end` variables in Algorithm 8.1, Line 1. The performance of PSIA with nonadaptive DLS techniques on homogeneous and heterogeneous computing systems is presented and analyzed below (Chapter 10).

**Algorithm 8.1 Spin-image calculation [EMC17a]**

adCalculateSpinImages (W, B, S, OP, M, spinImages, start, end)
**Inputs** : W: image width
B: bin size
S: support angle
OP: list of oriented points
M: number of oriented points
spinImages: list of spin-images to be filled

```
1  for imageCounter = start → end do
2      P = OP[imageCounter]
3      tempSpinImage[W, W]
4      init(tempSpinImage)
5      for j = 0 → M do
6          X = OP[j]
7          np_i = getNormal(P)
8          np_j = getNormal(X)
9          if acos(np_i · np_j) ≤ S then
10             k = ⌈ (W/2 − np_i · (X − P)) / B ⌉
11             l = ⌈ sqrt(||X − P||^2 − (np_i · (X − P))^2) / B ⌉
12             if 0 ≤ k < W and 0 ≤ l < W then
13                 tempSpinImage[k, l]++
14     add(spinImages, tempSpinImage)
```

### 8.1.2 Mandelbrot

The second application of interest is the Mandelbrot, which computes the Mandelbrot set [Man80] and generates its image. The application is parallelized such that the calculation of the value at every single pixel of a 2D image is a loop iteration that is performed in parallel.

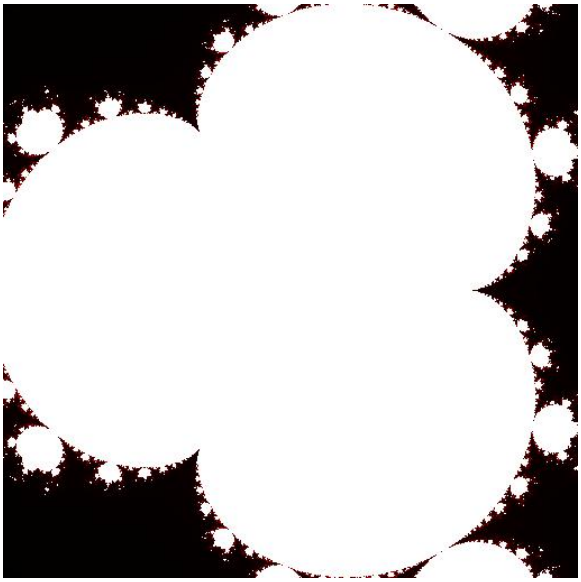


**Figure 8.3 Mandelbrot calculation at the seahorse valley for $z^4$.** White points represent high computational load due to several iterations to reach convergence, and black points represent negligible computations whereby saturation is reached in a few iterations.

Algorithm 8.2 shows the calculation of the Mandelbrot set for every pixel to generate a Mandelbrot set image. The for loop in Line 2 is parallelized with DLS techniques, and the values of *start* and *end* are calculated based on the chunk size obtained by a DLS technique. The application computes the function $f_c(z) = z^4 + c$ instead of $f_c(z) = z^2 + c$ to increase the number of computations per task (see Lines 10-12). Line 9 represents the primary source of load imbalance, as the number of repetitions of the calculations between Lines 9 to 14 is irregular. The size of the generated image is $512 \times 512$ pixels resulting in $2^{18}$ parallel loop iterations. The calculation is focused on the center image, where the computation is intensive to increase the variability between task execution times. Figure 8.3 shows the calculated image. Mandelbrot is often used to evaluate the performance of dynamic scheduling techniques due to the high variation between its loop iterations execution times.

**Algorithm 8.2 Mandelbrot set calculation**

**Inputs** : $W$: image width
$K$: max iterations
$RM$: real max
$Rm$: real min
$IM$: image max
$Im$:image min
$SR$: scale real
$SI$: scale image
$SC$:scale color
$data$: pixel information

```
1  N = 2
   /* start and end are set by the scheduling techinque
      according to the chunk size                          */
   /* calculate pixels in parallel                         */
2  for i = start → end do
3      z.real = z.imag = 0
4      rowID = i/W
5      colID = imodW
6      c.real = Rm + colID × SR
7      c.imag = Im + (W − 1 − rowID) × SI
8      k = 0 lengthsq = 0
9      while lengthsq < (N × N) do
10         temp = z.real^4 − 6 × z.imag^2 × z.real^2 + z.imag^4 + c.real
11         z.imag = 4 × z.real^3 × z.imag − 4 × z.real × z.imag^3 + c.imag
12         z.real = temp
13         lengthsq = z.real^2 + z.imag^2
14         k + +
15     data[i] = (k − 1) × SC
```

## 8.2 Computing Systems for Native and Simulative Experiments

Two HPC systems are used in the experiments included in this chapter. Comparing simulative and native performance results on two HPC systems show the usability of the proposed simulation methods (in Chapter 6) and confirms their accuracy as will be discussed below in Section 8.3.

### 8.2.1 The miniHPC system

The miniHPC[1] system is a high performance computing cluster of the Department of Mathematics and Computer Science at University of Basel, Switzerland. It consists of 26 compute nodes, a login node, and a storage node. The miniHPC cluster has a theoretical peak performance of 30 $TFLOP/s$. The first 22 compute nodes have two Intel Broadwell CPUs. Each node has a dual-socket Intel Broadwell $E5-2640$ v4 processor, with 20 cores (10 cores in each socket). Each core has 32 KB L1 instruction cache, 32 KB L1 data cache, and 256 KB L2 instruction and data cache. Each socket (10 cores) shares a 25 MB L3 cache. The total system memory is 64 GB distributed across the two non-uniform memory access (NUMA) domains (32 GB per socket). The four remaining compute nodes have standalone Intel Xeon Phi 7210 processors. Each node has 64 cores on the same NUMA node and contains 96 GB of main memory. Each core has 32 KB of L1 cache and a shared 32 MB of L2 cache.

The software and hardware characteristics of the *Broadwell partition* of the miniHPC system are listed in Table 8.1. All nodes are interconnected using the Intel Omni-Path interconnection fabric, in a nonblocking two-level fat-tree topology. The network bandwidth is 100 Gb/s, and the latency is 100 nanosecond. The miniHPC uses CentOS Linux release 7.2.1511 as the operating system, and the network file system version 4 (NFS4) is used as the file system for all the nodes.

### 8.2.2 The Taurus system

Taurus is a Bull HPC system at the Technische Universität Dresden, Germany. It comprises $2,085$ nodes with a total theoretical peak performance of $2,087$ $TFLOP/s$. For the experimental studies herein, 22 dual socket Intel Broadwell nodes are used. Each socket of theses nodes is equipped with an Intel Xeon CPU $E5-2680$ v4 running at $2.40GHz$ (Broadwell) a 14 cores. A node provides $64GB$ RAM. The nodes are connected via Infiniband FDR network, in a non-blocking full fat-tree topology. The peak bandwidth is $54.4Gbit/s$, and the latency is $0.7\mu s$. The software and hardware characteristics of the *Broadwell partition* of Taurus are listed in Table 8.1.

[1] miniHPC is a fully controlled research and teaching HPC cluster at the Department of Mathematics and Computer Science at the University of Basel, Switzerland, https://hpc.dmi.unibas.ch/HPC/miniHPC.html

**Table 8.1 The characteristics of the HPC systems**

| Parameter | Broadwell partition of miniHPC | Broadwell partition of Taurus |
|---|---|---|
| Operating system | CentOS Linux release 7.2.1511 | Red Hat Enterprise Linux Server release 6.9 |
| Job scheduler | Slurm v. 17.02.7 | Slurm v. 16.05.7 |
| MPI | Intel MPI v. 2017 update 1 | Intel MPI v. 2017 update 1 |
| File system | NFS4 | NFS4 |
| Number of nodes | 22 | 32 |
| Processor | Intel Xeon E5-2640 v4 | Intel Xeon E5-2680 v4 |
| Number of sockets | 2 | 2 |
| Cores per socket | 10 | 14 |
| Hyper-threading | enabled | disabled |
| Operating frequency | 2.4 – 3.4 *GHz* | 2.4 *GHz* |
| Peak performance per core | 38.4 – 54.4 *GFLOP/s* | 38.4 *GFLOP/s* |
| L1 cache | 32 *KB* per core | 32 *KB* per core |
| L2 cache | 256 *KB* per core | 256 *KB* per core |
| L3 cache | 25 *MB* per socket | 35 *MB* per socket |
| RAM | 64 *GB* per node | 64 *GB* per node |
| Topology | non-blocking fat tree | non-blocking fat tree |
| Interconnection | Intel Omni-Path | Inifiniband FDR |
| Bandwidth | 100 *Gbit/s* | 54.4 *Gbit/s* |
| Latency | 100 *ns* | 700 *ns* |

## 8.3 Experimental Verification Results

Three sets of experiments are presented herein. Each set of experiments test and compare different simulators or different methods to represent the computational effort in a task on two different HPC systems. The results of each set of experiments are compared to measure the accuracy of simulation quantitatively and qualitatively.

### 8.3.1 Two Systems, Two Simulations Interfaces per System

The first set of experiments is to compare two simulators in predicting the performance of an application on the two HPC systems presented above (see Section 8.2). The difference between native and simulative performance results are measured to evaluate how realistic are performance simulations quantitatively.

**Table 8.2 Two HPC systems and two simulators experiments details**

| Factors | Values | Properties | |
|---|---|---|---|
| **Application** | PSIA | N = 400,000 tasks | |
| **Loop Scheduling** | STATIC | Static | |
| | SS<br>FSC<br>GSS<br>FAC | Nonadaptive dynamic | |
| **Computing systems** | miniHPC | 22 nodes × 16 cores = 352 cores | |
| | Taurus | 22 nodes × 16 cores = 352 cores | |
| **Experimentation** | Native | On miniHPC | |
| | | On Taurus | |
| | Simulative | Using SG-MSG | Simulate miniHPC |
| | | | Simulate Taurus |
| | | using SG-SD | Simulate miniHPC |
| | | | Simulate Taurus |

#### 8.3.1.1 Experiments and setup

Table 8.2 summarizes the experiments performed in this subsection. In this set of experiments, SG-MSG and SG-SD are used to simulate the performance of PSIA on both, miniHPC and Taurus. The PSIA is parallelized using MPI, and each MPI rank is pinned to one processing core in miniHPC or Taurus. DLS techniques are implemented using a decentralized coordination approach [EC19a]. The two simulation methods (SG-MSG and SG-SD) represent two different methods to represent the application as a data-parallel application (SG-MSG) or a task-parallel.

The amount of work contained in each task is measured using PAPI [BDG+00]. The FLOP count obtained with PAPI is used to represent the amount of work in each task in SG-MSG/SG-SD. Both miniHPC and Taurus are represented in SG by `platform files` as by the method described in Section 6.3. The core speed is calculated by measuring the loop execution time in a sequential run to avoid any parallelization or communication overhead. The sum of the total number of FLOP in all iterations is divided by the measured loop execution time to estimate the core processing speed (see Section 6.3).

#### 8.3.1.2 Results and discussion

The results of the simulative executions with SG-MSG and SG-SD compared to the native execution on both, miniHPC and Taurus, are shown in Figure 8.4. The figure shows the percent error $\%E$ calculated using Equation 7.1. The results show that, in general, the simulative execution tends to underestimate the execution times. However, SG-MSG simulation tends to overestimate the execution time on miniHPC and underestimate the execution time on Taurus. On the contrary, SG-SD always underestimates the execution time on both HPC systems.

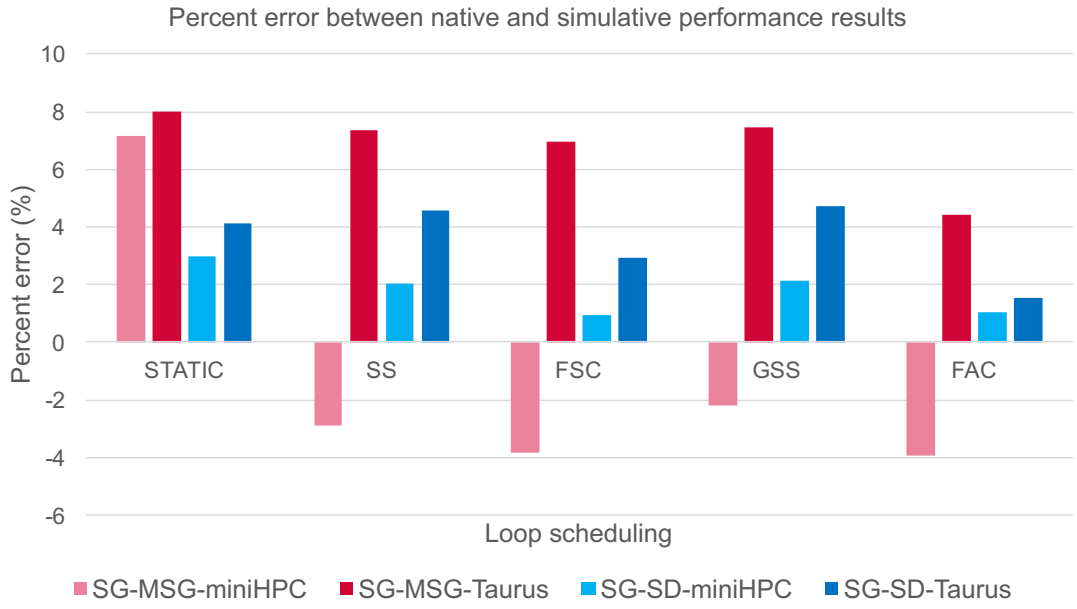
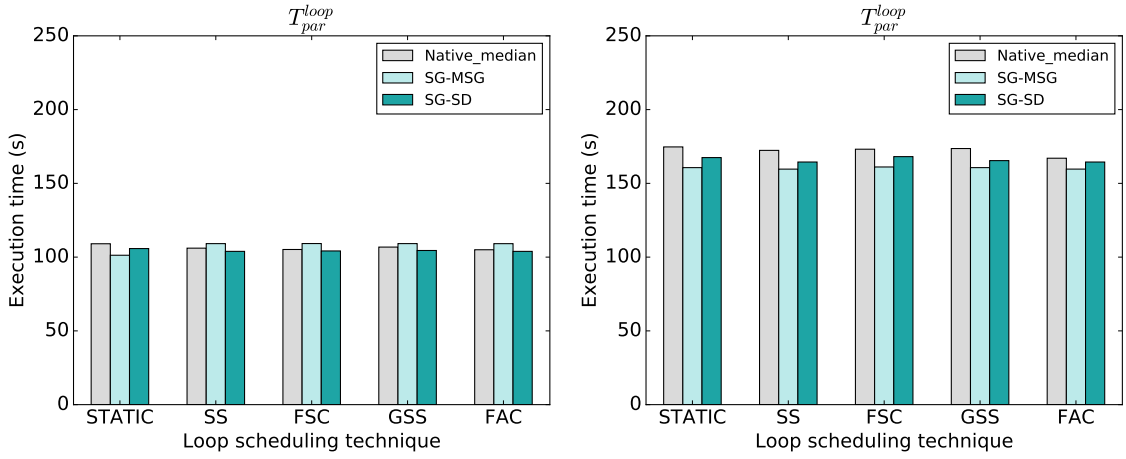



**Figure 8.4 Comparison between native and simulative performance with different simulation approaches.** The percent error $\%E$ between native and simulative (SG-MSG and SG-SD) parallel loop execution times $T_{par}^{loop}$ of the PSIA on miniHPC and Taurus [MEC+18a].

The median of the $\%E$ of the SG-MSG simulations is $-2.89\%$ compared to $2.05\%$ with the SG-SD simulation for the miniHPC execution. For the execution on Taurus, the medians of the $\%E$ are $7.37\%$ and $4.14\%$ for SG-MSG and SG-SD, respectively. The results of both simulation interfaces (SG-MSG and SG-SD) tend to underestimate the execution time on Taurus for scheduling techniques that incur high overhead, such as SS, as both simulations do not fully capture the scheduling overhead. For example, SG-MSG only accounts for the messages to send the chunk size, whereas SG-SD considers the FLOP count of the chunk calculation and the messages to send the chunk size. *This shows the importance of simulating the details of DLS techniques and their impact on the predicted performance.* For the execution on miniHPC, both the SG-MSG and the SG-SD simulations correctly predict that STATIC results in the worst performance and that FAC

outperforms all other techniques, similar to the native execution. For the execution of Taurus, both simulators correctly predict that FAC outperforms other loop scheduling techniques. Both simulations incorrectly predict that the worst performance occurs with FSC, instead of STATIC as the native execution. However, given that PSIA is slightly load-imbalanced, the performance of all DLS techniques considered herein (STATIC, SS, FSC, GSS, and FAC) is very close to each other. PSIA performance with DLS techniques represents a corner case for the performance prediction using simulation to capture and reproduce such small performance differences between different DLS techniques. *For the selection of a scheduling technique, simulation (with either SG-MSG or SG-SD) correctly predicts the best DLS technique.*




(a) Parallel loop exeuciton time (miniHPC)

(b) Parallel loop exeuciton time (Taurus)

**Figure 8.5 Simulative performance (in terms of $T_{par}^{loop}$) of the PSIA using DLS obtained using SG-MSG and SG-SD.** Simulative execution results are compared with the median of 20 repetitions of the native executions results [MEC+18a].

## 8.3.2 Time Vs. FLOP Count

In this set of experiments, we investigate the effect of using time measurements and FLOP count to represent the computational effort per task on the predicted simulative performance.

### 8.3.2.1 Experiments and setup

We perform a subset of the experiments described above in Section 8.3.1 for this test. Table 8.3 lists the details of the performed experiments. Only one HPC system is used in this experiment, and only SG-SD simulation is used. Two methods are used to represent the computational effort per task in simulation: (1) Time and (2) FLOP count. The

**Table 8.3 Time vs FLOP count experiments details**

| Factors | Values | Properties | |
|---|---|---|---|
| **Application** | PSIA | N = 400, 000 tasks | |
| **Loop Scheduling** | STATIC | Static | |
| | SS<br>FSC<br>GSS<br>FAC | Nonadaptive dynamic | |
| **Computing systems** | miniHPC | 22 nodes × 16 cores = 352 cores | |
| **Experimentation** | Native | On miniHPC | |
| | Simulative | Using SG-SD | Using task execution time |
| | | | Using FLOP count per task |

native PSIA code is instrumented with timers at the start and end of each loop iteration (task) to measure a task execution time. `MPI_Wtime` timing function is used as the MPI standard specifies that it incurs the lowest possible measurement overhead. The application is executed in a sequential run to avoid any parallelization/scheduling overhead and task execution times were recorded and written to a file. These task execution times are multiplied by the core speed in the `platform file` to convert time to FLOP count and are fed later to the SG-SD simulator. The FLOP count was measured using the method described above (see Section 8.3.1).

#### 8.3.2.2 Results and discussion

The percent error $\%E$ is calculated between native and simulative performance results, as shown in Figure 8.6. The results show that using time measurements for the representation of the tasks results in a high overestimation of the parallel loop execution time $T_{par}^{loop}$. The overhead of time measurement dominates the measured task execution time. This inaccurate time measurement may be attributed to the fine granularity of PSIA tasks, where a task execution time is around 2 *ms*. *The time-based measurements could not capture the dynamic behavior of the application and is affected by the measurement process.*

Representing the computational effort in an application using the FLOP count is more accurate. *FLOP count represents the amount of work regardless of the platform computing speed as opposed to time measurements. Also, FLOP count can be accurately measured even at the fine-grained tasks.* The time measurement, however, can be used at the gross grain of the loop execution time to estimate the core speed, as described in

Section 6.3.

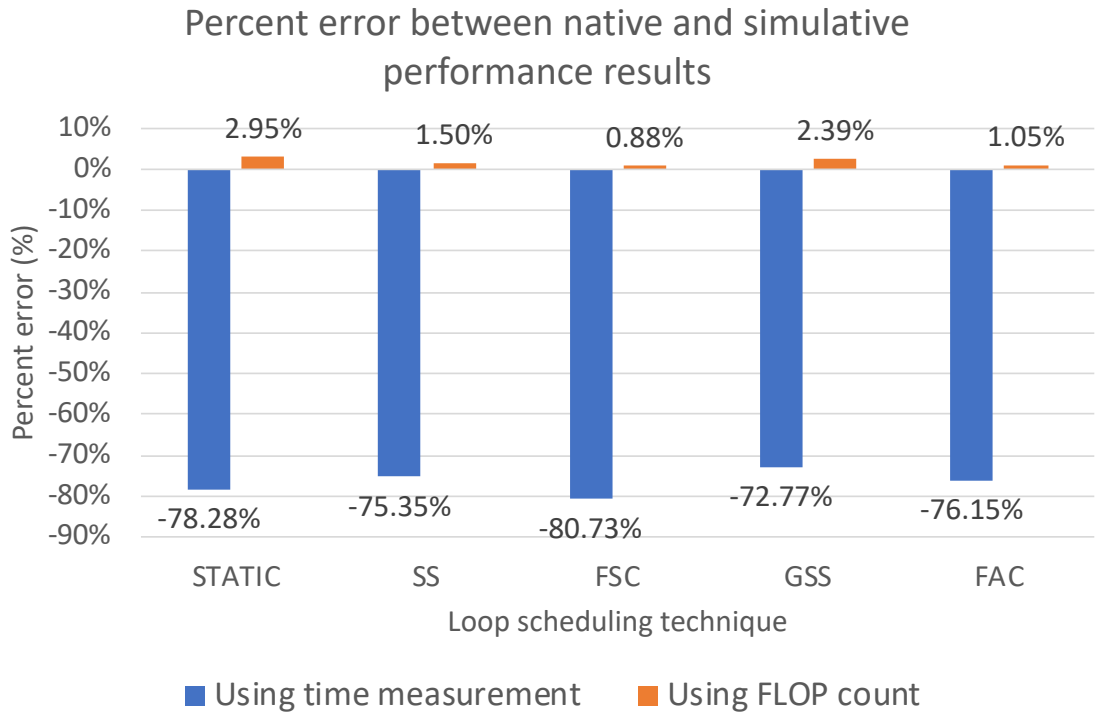



**Figure 8.6 Comparison between using time and FLOP count to represent workload per task.** The percent error $\%E$ between native and simulative SG-SD parallel loop execution times $T_{par}^{loop}$ of the PSIA on miniHPC using time and FLOP count for task representation.

### 8.3.3 FLOP Vs. FLOP Distribution

In this set of experiments, we explore representing tasks using FLOP count and by drawing FLOP count per task from a probability distribution that represents the FLOP count distribution of a particular application. FLOP count distribution (*FLOP_dist* hereafter) is proposed to capture the dynamic nature of native experimentation between several executions of the same experiment.

#### 8.3.3.1 Experiments and setup

Two applications are used in this experiment set, PSIA and Mandelbrot. Using Mandelbrot, as a highly imbalanced application, allows the evaluation of how realistic simulations using SG captures the native performance characteristics. Applications are parallelized using MPI, and SMPI+MSG simulation approach is used to simulate applications' performance. Static, dynamic nonadaptive, and adaptive DLS techniques are used to

load balance the execution of the two applications on miniHPC. The *DLS4LB* (see Chapter 9) is used to employ DLS to the applications of interest. Table 8.4 summarizes the details of these experiments.

**Table 8.4 FLOP count vs *FLOP_dist* experiment details**

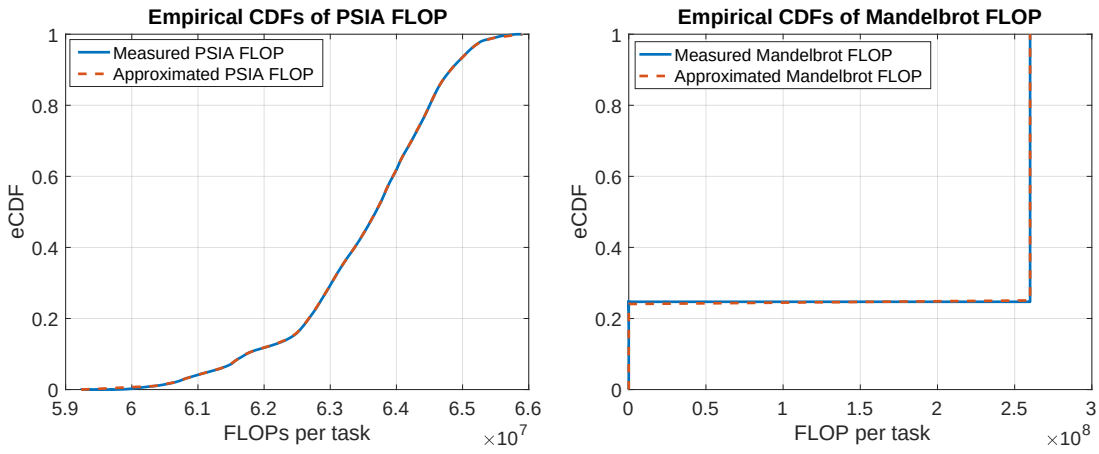
| Factors | Values | Properties |
|---|---|---|
| **Applications** | PSIA | $N = 400,000$ tasks<br>Low variability among tasks |
| | Mandelbrot | $N = 262,144$ tasks<br>High variability among tasks |
| **Self-scheduling techniques** | STATIC | Static |
| | mFSC, GSS, FAC | Dynamic nonadaptive |
| | AWF-B, -C, -D, -E | Dynamic adaptive |
| **Computing system** | miniHPC | 16 Dual socket Intel $E5-2640v4$ nodes<br>10 cores per socket<br>64 GB memory<br>Nonblocking fat-tree topology |
| **Experimentation** | Native | $P = 256$ miniHPC PEs, using 16 nodes, 16 PE per node |
| | Simulative | $P = 256$ simulated miniHPC PEs, 16 nodes, 16 PE per node<br>(1) Using *FLOP file* with SG-SMPI+SG-MSG<br>(2) Using *FLOP distribution* with SG-SMPI+SG-MSG |

Two methods are experimented to represent applications tasks in simulation:

1. *FLOP_file*
2. *FLOP_dist*

The FLOP count per task was measured using the method described above (see Section 8.3.1). The FLOP counts are then read from a file during simulation, denoted *FLOP_file*. For the *FLOP_dist*, a probability distribution is fitted to the measured FLOP count to simulate the dynamic behavior of the task execution times. The linear piecewise approximation of the empirical cumulative density function (eCDF) is used [BEDG19] to obtain this probability distribution. The eCDF values are split over the y-axis into 100 linear segments (pieces). A segment is randomly selected to draw a sample from this distribution, and a value is randomly selected along this linear segment. Figure 8.7 shows the results of approximating the measured FLOP counts of tasks both from PSIA and Mandelbrot using linear piecewise approximation of the eCDF using MATLAB[2]. To ensure

[2] https://www.mathworks.com/products/matlab.html




(a) eCDF of PSIA tasks FLOP count

(b) eCDF of Mandelbrot tasks FLOP count

**Figure 8.7 The empirical cumulative density function of the tasks FLOP counts of PSIA and Mandelbrot.** The distribution of the measured FLOP count is shown in blue, and the distribution of the FLOP counts drawn from the linear piecewise approximation of the eCDF is shown in orange. The results show that approximated distribution represents the empirical measured FLOP counts of both applications closely [MEC+19].

that the simulator draws samples from the approximated distribution with a fast, long period, and low serial correlation random engine, the random number generator of the GNU Scientific Library[3] (GSL) is used in the simulator to generate good uniformly distributed random numbers to select among the 100 linear segments and a value from the segment with low overhead during simulation. In *FLOP_dist*, the simulation is repeated 20 times similar to the native execution with different seeds to capture the variability of the performance of the native application.

#### 8.3.3.2 Results and discussion

Figures 8.8 to 8.10 show the native, SMPI+MSG simulative with *FLOP_file*, and SMPI+MSG simulative with *FLOP_dist* performance of both PSIA and Mandelbrot with eight static and dynamic (nonadaptive and adaptive) self-scheduling techniques. Each experiment is repeated 20 times to obtain performance results with high confidence. The boxes represent the first and third quartiles, the red line represents the median of the 20 measurements, and the whiskers represent 1.5× the standard deviation of the results. Parallel loop execution time $T_{par}^{loop}$ is used to measure applications' performance. Also, two metrics (see Section 2.1), namely the coefficient of variation of PEs finishing times (c.o.v.) and the mean of PEs finishing times divided by their maximum value (mean/max),

[3] https://www.gnu.org/software/gsl/doc/html/index.html

are used to measure the load imbalance.

Several performance features are extracted by analyzing the performance of the two applications with DLS techniques. STATIC degraded the performance of both PSIA and Mandelbrot due to load imbalance. The high values of c.o.v and max/mean in both applications indicate the load imbalance with STATIC as shown in Figures 8.8(c) and 8.8(d). Although the value of c.o.v for GSS is lower than that of mFSC for PSIA, one can see that the performance of GSS is worse than mFSC. Figure 8.8(e) shows, however, that the value of max/mean for GSS is higher than that of mFSC, which explains the large execution time in Figure 8.8(a). PSIA performance with GSS is an example where the c.o.v. hides the load imbalance resulting from a single process lagging the application execution, as explained above. FAC technique improves the performance of both applications and result in the lowest execution time and also load imbalance metrics. The adaptive techniques improve the performance of PSIA and result in low load imbalance metrics as well. However, for the Mandelbrot due to the high variability of its tasks execution times and short application execution time, the adaptive techniques did not have enough time to estimate PE relative weights correctly and resulted in high execution time and high load imbalance metric values with high variability also.

The native and simulative performance of PSIA and Mandelbrot is analyzed in terms of $T_{par}^{loop}$, c.o.v., and max/mean metrics to evaluate how realistic are the performed simulations. *Realistic simulation results are expected to lead to a similar analysis and similar conclusions drawn from the analysis of the native results.* Table 8.5 summarizes seven performance features extracted from the analysis of applications' performance with various scheduling techniques. The comparison between the native and simulative performance analysis shows that the simulations with *FLOP_file* captured almost all the performance features that characterize the performance of the two applications under test. The simulator overestimated only the performance of AWF-B and AWF-D.

**Table 8.5 Native application performance features realistically captured by simulations**

| Performance features | SMPI+MSG *FLOP_file* | SMPI+MSG *FLOP_dist.* |
|---|---|---|
| Load imbalance with STATIC (PSIA, Mandelbrot) | Captured | Not captured |
| High c.o.v. with mFSC (PSIA) | Captured | Captured |
| Long $T_{par}^{loop}$, low c.o.v., and high max/mean with GSS (PSIA) | Captured | Captured |
| FAC best performance (PSIA, Mandelbrot) | Captured | Captured |
| Adaptive techniques high performance (PSIA) | Partially captured | Partially captured |
| Adaptive techniques poor performance (Mandelbrot) | Captured | Captured |
| Adaptive techniques high variability (Mandelbrot) | Not captured | Not captured |

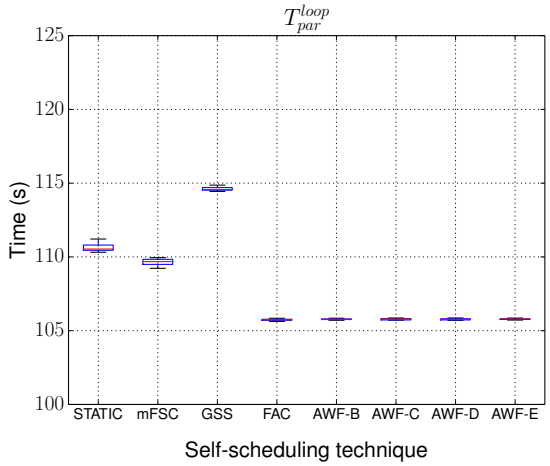
Both simulations predicted correctly that the FAC technique achieves a balanced load execution for both applications and improves performance. Simulations with the *FLOP_dist* failed to capture the load imbalance with STATIC in both applications. The performance with STATIC is significantly affected by the order of the tasks or the loop iterations assigned to each PE. As the order of tasks is not preserved by drawing random samples from FLOP distributions, the load imbalance with STATIC is dissolved between PEs as they are assigned different tasks in simulative execution from the native one.

Interestingly, both simulations were able to capture the most devious performance feature of high $T_{par}^{loop}$, low c.o.v, and high max/mean values of GSS with PSIA. Both simulations did not capture the high variability of adaptive techniques. The adaptive techniques depend on time measurements to estimate PE performance. If the granularity of the tasks is highly variable and some task sizes are very fine, the time measurement of their execution will be inaccurate due to overhead of the time measurement (probing effect). The inaccurate time measurement leads to incorrect weight estimation and high variability between different native executions. This probing effect does not exist in the simulative execution and, therefore, was not fully captured. This explains the overestimation of execution time with AWF-B and AWF-D mentioned above. However, both simulations correctly predicted the high performance of adaptive techniques with PSIA and their low performance with Mandelbrot. The simulation with *FLOP_dist* was able to capture the small variability in performance with various DLS techniques, which was not captured by reading the FLOP counts from a file in the first simulation.

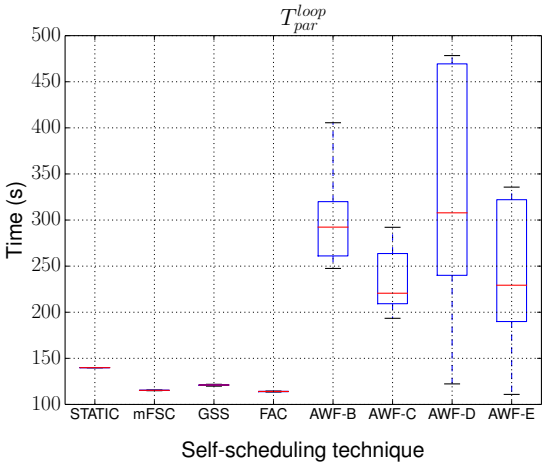
## 8.4 Discussion

*Realistic simulation results lead to a similar analysis and conclusions to the analysis of the native results.* The accuracy of simulative performance was examined using several experiment sets. Using FLOP count per task (either reading the exact values from a file or drawing from a probability distribution) to represent the computational effort per task in simulation is found to be more accurate. Simulations with FLOP count produced performance predictions within 8% (with SG-MSG) and 5% (with SG-SD) of the native performance results. Simulation always correctly predicted the most efficient DLS technique in comparison with other DLS techniques. The comparison of performance analysis from native and simulative performance results shows that simulation fully captured most of the performance characteristics of interest. Simulation captured even complex performance characteristics, such as the case of PSIA performance with GSS. *Therefore, applying the simulation method presented above (Chapter 6) achieved realistic simulations of scientific applications performance.* Realistic simulations of performance are used in

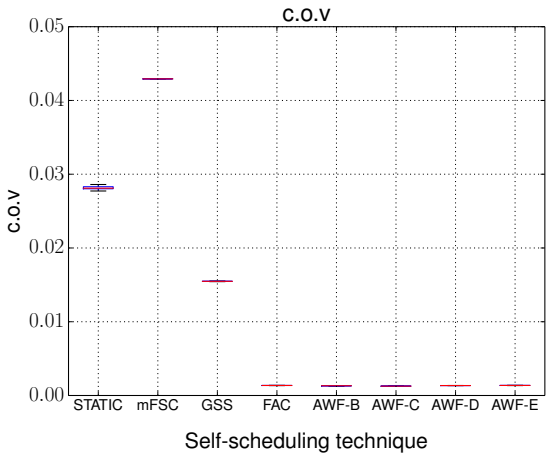
the next chapters to predict applications' performance with different DLS techniques under various execution scenarios.



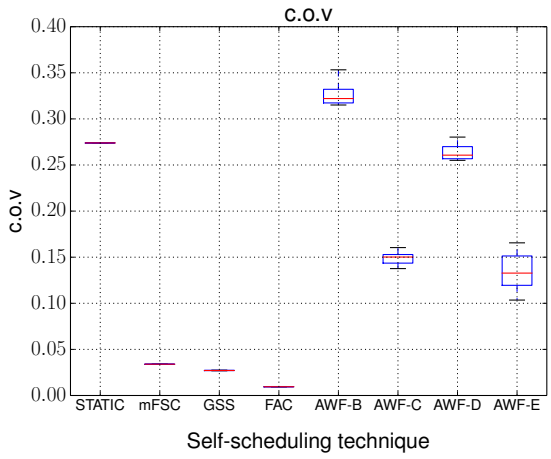

(a) PSIA native performance



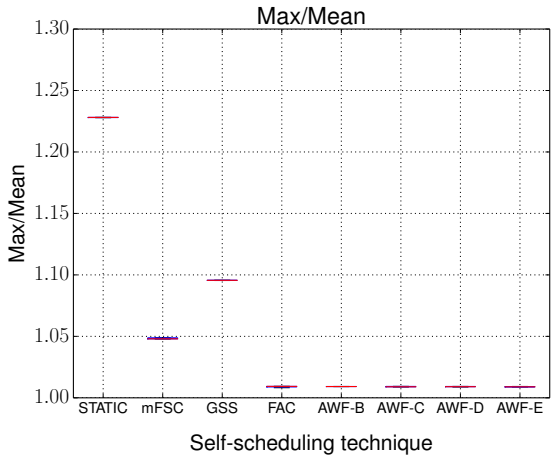

(b) Mandelbrot native performance



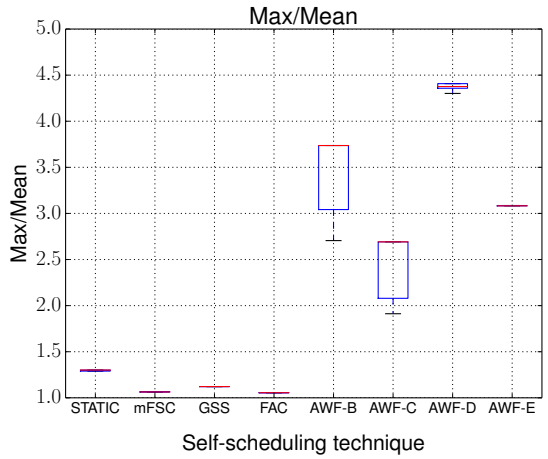

(c) Load imbalance in PSIA (c.o.v.)




(d) Load imbalance in Mandelbrot (c.o.v.)

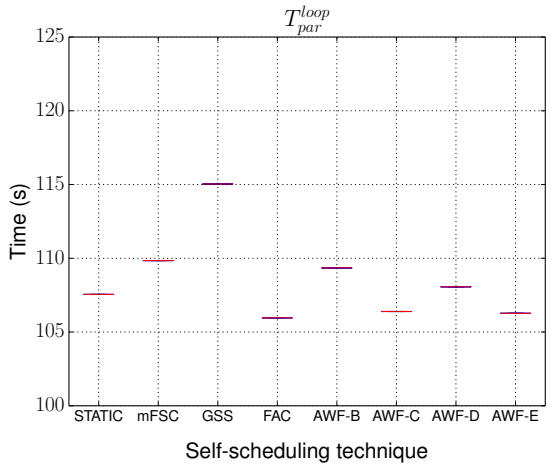



(e) Load imbalance in PSIA (max/mean)

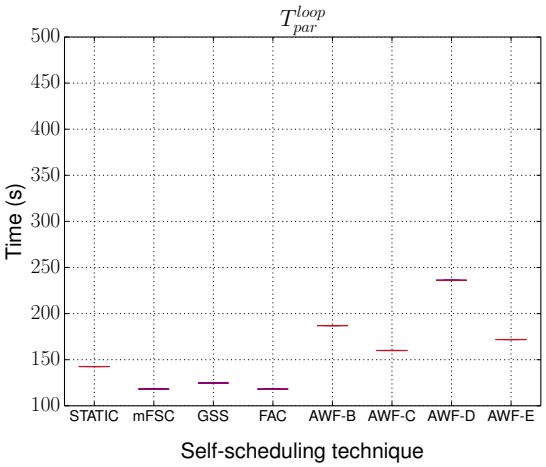



(f) Load imbalance in Mandelbrot (max/mean)

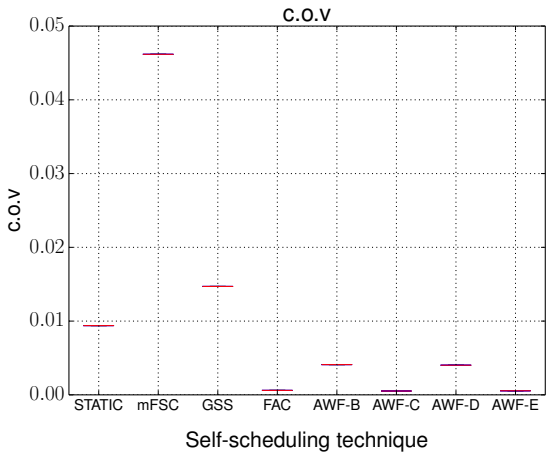
**Figure 8.8 Native performance of PSIA and Mandelbrot applications.** STATIC degrades applications performance due to high load imbalance. Applications performance improves with FAC. Adaptive techniques improve the performance of PSIA; however, they degrade Mandelbrot performance and do not adapt correctly [MEC+19].



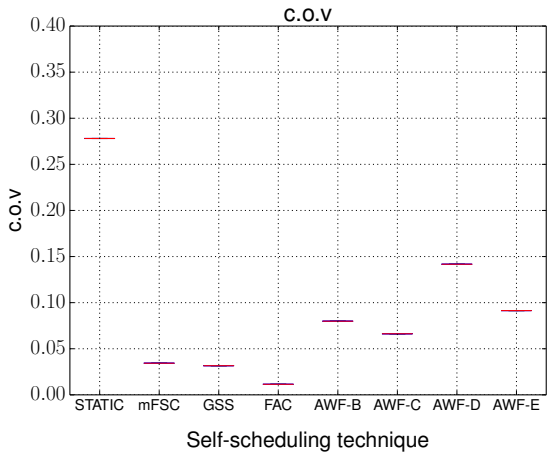

(a) PSIA simulative performance



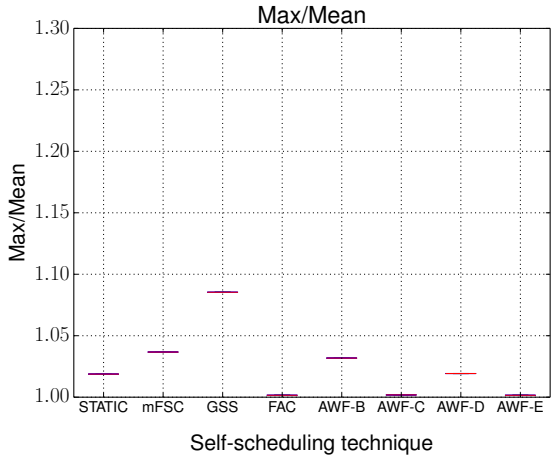

(b) Mandelbrot simulative performance



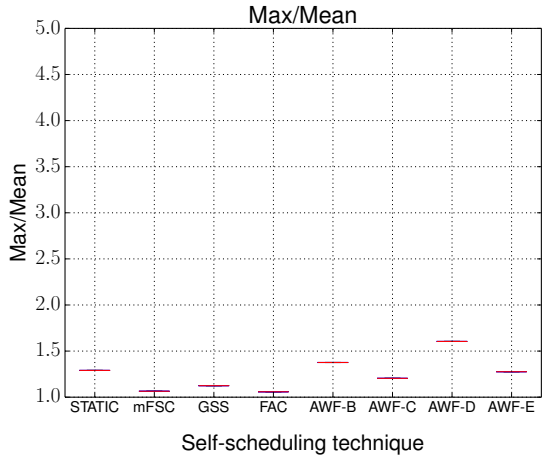

(c) Load imbalance in PSIA (c.o.v.)




(d) Load imbalance in Mandelbrot (c.o.v.)

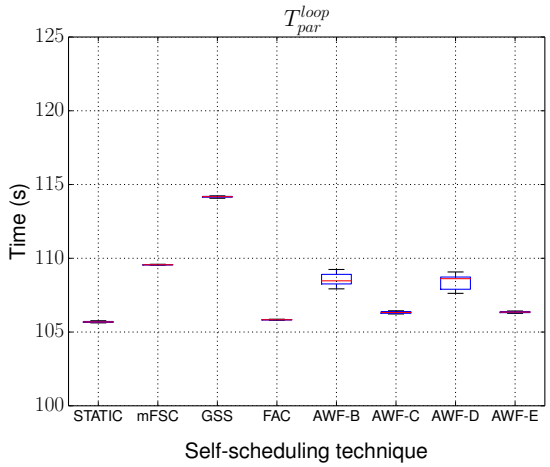



(e) Load imbalance in PSIA (max/mean)

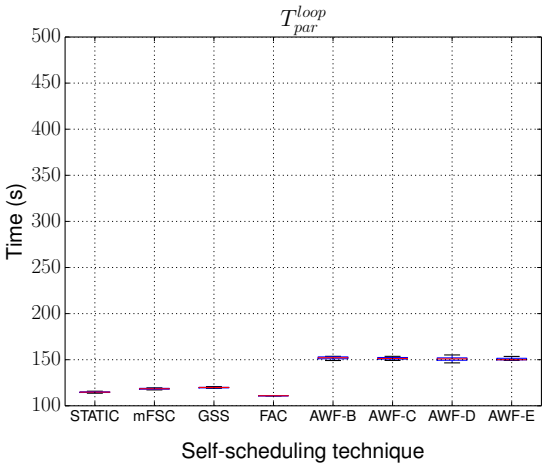



(f) Load imbalance in Mandelbrot (max/mean)

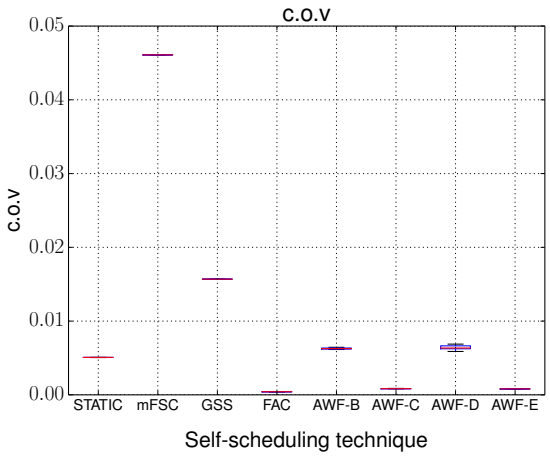
**Figure 8.9 Simulative performance of PSIA and Mandelbrot applications with reading *FLOP_file*.** STATIC results in imbalanced load execution for PSIA and Mandelbrot and degrades the performance. GSS results in poor PSIA performance due to a process lagging the execution. FAC improves the performance of both application via a balanced load execution. Adaptive techniques result in enhanced PSIA performance and poor Mandelbrot performance [MEC+19].



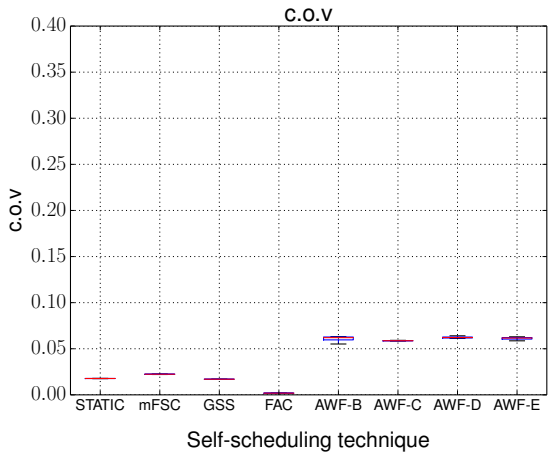

(a) PSIA simulative performance



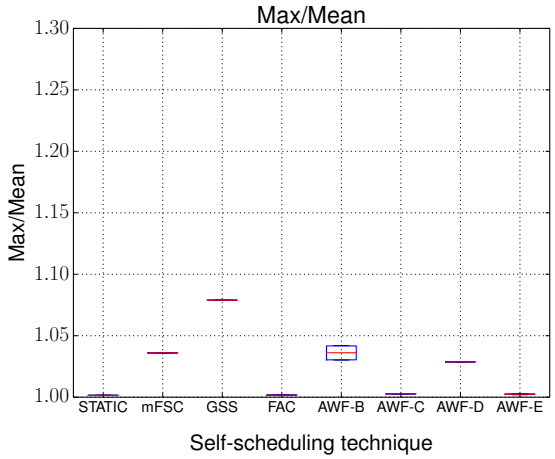

(b) Mandelbrot simulative performance



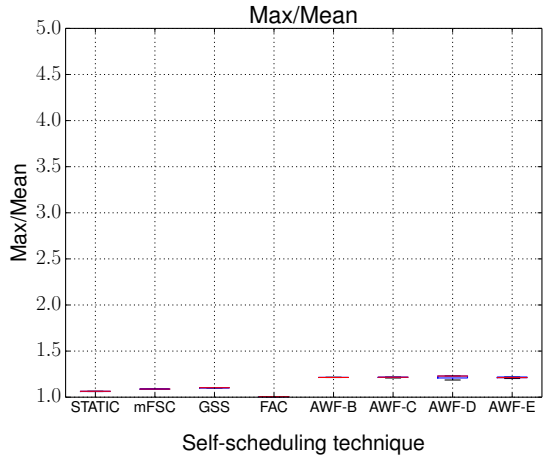

(c) Load imbalance in PSIA (c.o.v.)




(d) Load imbalance in Mandelbrot (c.o.v.)




(e) Load imbalance in PSIA (max/mean)




(f) Load imbalance in Mandelbrot (max/mean)

**Figure 8.10 Simulative performance of PSIA and Mandelbrot applications with *FLOP distribution*.** STATIC, FAC, AWF-C, AWF-E results in the best PSIA performance. GSS degrades PSIA performance and mFSC results in high load imbalance. FAC achieves the best performance for both applications. Adaptive techniques degrade Mandelbrot performance [MEC+19].

Part III

# Load Balancing of Scientific Applications

# 9

# Implementation of Dynamic Loop Scheduling for Applications in Shared and Distributed Memory Systems

Modern HPC systems are increasing in the hardware parallelism they offer within a compute node, by increasing the number of PEs per node, and across nodes, by increasing the number of nodes per system. Scientific applications need, therefore, to harness this hardware parallelism by multithreaded, multiprocess, or hybrid (multithreaded multiprocess) programming approaches. OpenMP and MPI are the de-facto standards of multithreaded and multiprocess programming in HPC systems. In this chapter, we show how verified DLS techniques are implemented in OpenMP and MPI (see Figure 9.1).

Implementation and support of the most advanced load balancing methods, such as DLS, to well-known and wide-spread programming models, such as OpenMP and MPI, enable scientific applications to benefit and improve their performance with minimal code changes. Providing implementations of DLS techniques encourages scientific applications' programmers to adopt advanced load balancing methods required for future large-scale irregular HPC systems.

## 9.1 DLS Implementation in OpenMP

In this section, we discuss the extension of the OpenMP GNU runtime library with DLS techniques in detail. Also, we highlight other similar efforts to extend OpenMP with more scheduling techniques.

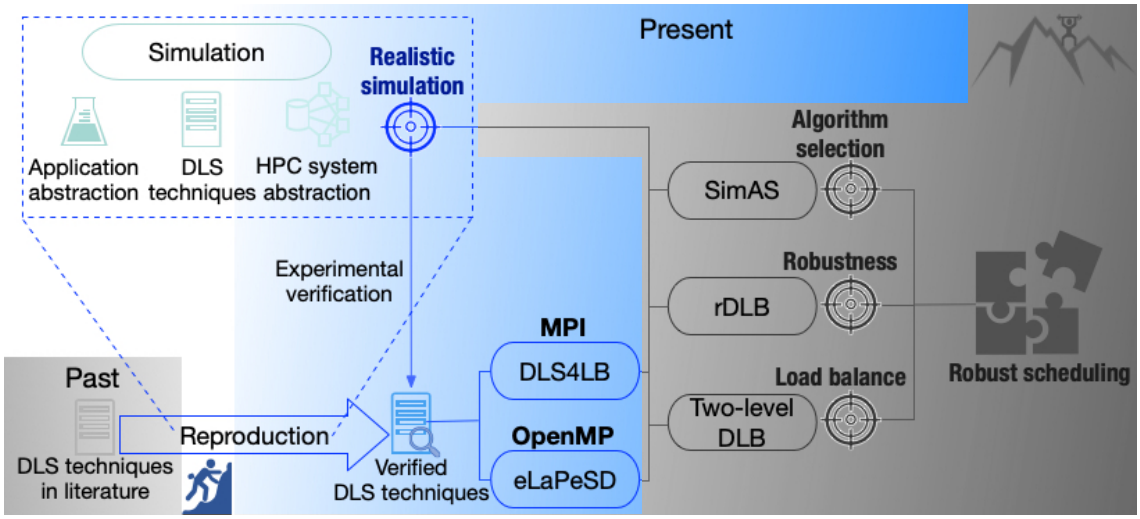



**Figure 9.1 Illustration of the focus and the progress towards robust scheduling (in colors) as part of the overall approach (in grayscale).** Verified DLS are implemented in OpenMP (*eLaPeSD*) and MPI (*DLS4LB*). Verified DLS implementations are used in the next chapters for the dynamic load balancing of scientific applications at a single and two levels of parallelism.

### 9.1.1 Extension of the OpenMP GNU Runtime Library with DLS

The work-sharing OpenMP construct for parallelizing a *for* loop is the most common OpenMP construct that is found in scientific applications.

*#pragma omp parallel for*

A *schedule(type, size)* clause can be added to the above *for* construct to specify the loop scheduling technique. Originally, the OpenMP standard specifies three loop scheduling techniques, namely *static*, *dynamic*, and *guided*, which are equivalent to STATIC, SS, and GSS described in Section 2.2 if used without specifying a *size*. The *size* in the schedule clause can modify the chunk sizes calculated by these techniques. In static and dynamic, OpenMP assigns a fixed-size chunk equals to *size*, either statically or dynamically, respectively. With *guided* schedule, the *size* is the minimum chunk size of GSS.

In addition to the three scheduling techniques, two other options can be provided as a *schedule(type)*: *auto* and *runtime*. With *auto*, the compiler automatically chooses one of the three techniques (*static*, *dynamic*, and *guided*) which it assumes would improve the performance. The method by which the compiler choose a scheduling technique is left to the implementation. *Runtime* option allows the user to select a scheduling technique from the techniques available in the OpenMP runtime library during execution. We note that OpenMP is a standard specification for parallel programming and compilers implement these specifications. Compilers, such as GNU, Intel, Cray, and LLVM, all implement OpenMP standard. We also note that there might be differences between

various compilers in their OpenMP support as long as these differences do not conflict with the OpenMP specification. For example, the LLVM compiler supports TSS as a fourth scheduling technique besides STATIC, SS, and GSS.

Two approaches can be employed to extend scheduling techniques in OpenMP:

1. Extend the compiler.

2. Extend the runtime library.

Extending the runtime library is found to be more suitable as it does not require the re-compilation of applications to change the used scheduling technique. As will be shown below (Chapter 12), the best performing DLS technique is not fixed and changes by changing execution parameters, such as problem size, system size, and perturbations in the execution. Therefore, changing the DLS technique during runtime is a significant advantage. Also, adding DLS techniques to the compiler might interfere with compiler optimizations and produce a less quality code. Therefore, the second approach of extending the runtime library is selected.

Four major changes are needed to define a new scheduling technique in OpenMP runtime library, namely:

1. The addition of the new scheduling technique such that it is recognized by *omp_schedule* environment variable

2. The definition of an initialization function

3. The definition of a next function that calculates and allocates a new chunk for the calling thread

4. The definition of a finish function that is called when the loop ends.

Figure 9.2 shows these four components.

In the initialization and finalization functions, all pre- and post-calculations that are required by a DLS technique are performed. For instance, the calculation of the fixed-rate by which the chunk size is decreasing delta $D$ in Equation (2.10) is performed in the initialization step. The next function is the most critical part, as it is called frequently by all threads when they become free to calculate and allocate a new chunk. The equations of chunk size calculations in Section 2.2 are performed herein. Care must be taken to avoid data races, as multiple threads call this function in parallel. Modifications to shared variables, such as the current loop index or the number of remaining loop iterations must be performed using atomic operations. We note that OpenMP scheduling following a decentralized coordination approach, where each thread calculates and allocates a new

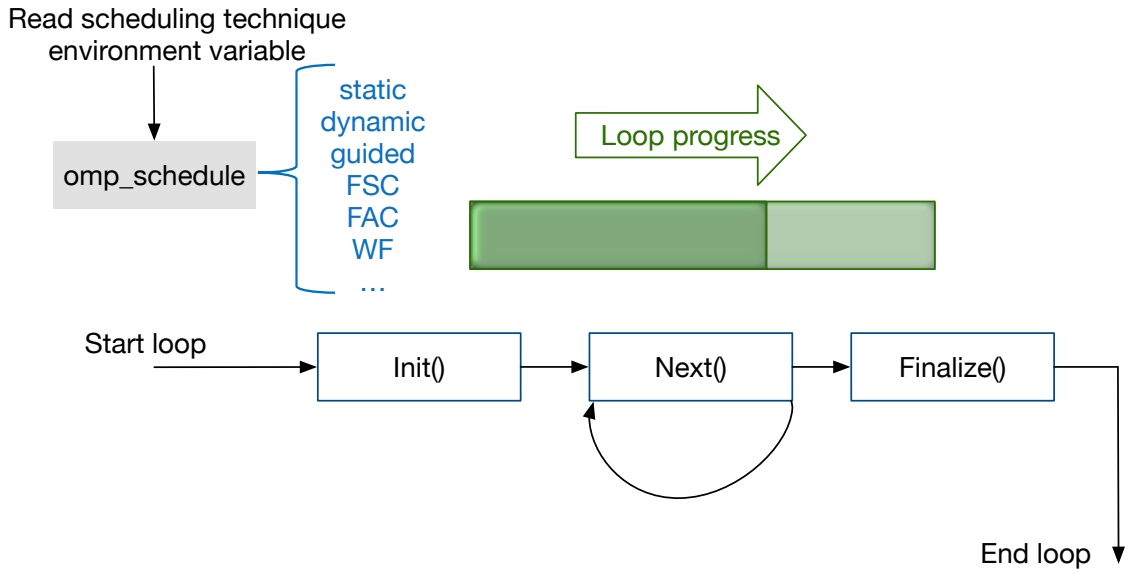



**Figure 9.2 An overview of the major components of the OpenMP runtime library that need to be modified to add more scheduling techniques.**

chunk of iteration for itself as soon as it becomes free, similar to the approach illustrated above in Figure 7.3(b).

The GNU runtime library, namely LaPeSD libGOMP [Lap], is extended into *eLaPeSD* [Bud17] with seven additional DLS techniques, namely: FSC, TSS, FAC, WF, Taper [Luc92] (TAP), bold strategy [Hag97] (BOLD), and RAND. Experiments with *eLaPeSD* in scheduling loops from Rodinia [CBM+09], OmpSCR [DRS05], NAS [Bai11], and SPEComp [ADE+01] benchmarks show that DLS techniques improves applications performance and more scheduling options are needed in OpenMP [CIB18].

### 9.1.2 Other OpenMP Extensions

Similar to the OpenMP runtime library approach presented above, Smart Round-Robin [PIC+17] (SRR) was implemented in the GNU OpenMP runtime library. SRR considers the task execution times to achieve near-optimal load balance. As such, it requires the profiling of an application before execution. BinLPT [PCP+17], another loop scheduling method, is also implemented in the GNU OpenMP runtime library. BinLPT is similar to SRR, except that it schedules chunks of loop iterations similar to DLS techniques, instead of single loop iterations in SRR.

Other OpenMP runtime libraries, such as the LLVM OpenMP runtime library, was extended as well. The LLVM runtime library was extended with the FAC technique [KTV+19]. Also, *static_steal*, which was created by mixing static and dynamic OpenMP scheduling, was used to extend the LLVM runtime library [KRG14]. *static_steal* has the advantage of data locality similar to static and dynamically balance the

load at the end of the execution using work-stealing. DLS techniques (nonadaptive and adaptive) was also implemented in the LLVM OpenMP runtime library [Yil19], namely FSC, TAP, FAC, WF, AWF-B, AWF-C, AWF-D, AWF-E, BOLD, and AF.

In addition to implementing scheduling techniques in OpenMP runtime libraries, the support for user-defined scheduling (UDS) was proposed to the OpenMP to be a part of the standard and, therefore, be supported by various compilers and runtime libraries [KIK+19]. Also, an interface was implemented in the runtime library to allow user-defined scheduling policies in the LLVM runtime library [BGB+19]. As such, it allows users to create and use their scheduling techniques that best suit their applications.

## 9.2 DLS Implementation in MPI

As described in Section 7.2.2 above, DLS techniques can be implemented using a centralized or decentralized coordination approach. We present here two MPI-based implementations of DLS techniques that cover the two implementation approaches.

### 9.2.1 Centralized Coordination

A dynamic load balancing tool (*DLB_tool*) [CB07] was introduced. The *DLB_tool* is implemented in C and FORTRAN programming languages to support scientific applications. The *DLB_tool* parallelizes and load balances scientific applications that contain simple parallel loops (1D loops) or nested parallel loops (2D loops). The tool employs a master-worker model where workers request work from the master whenever they become free. The master serves work requests and assigns workers chunks of loop iterations according to the selected DLS technique. Figure 9.3 shows the employed master-worker scheduling approach employed in the *DLB_tool*. The master also doubles as a worker and executes chunks of loop iterations when it is not serving any requests. The *DLB_tool* supports nine scheduling techniques, namely STATIC, mFSC, GSS, FAC, AWF-B, AWF-C, AWF-D, AWF-E, and AF.

The *DLB_tool* was designed to load balance scientific applications with minimum code changes. Algorithm 9.1 shows the changes needed to use the *DLB_tool* in a scientific application in blue font color. Descriptions of the *DLB_tool* functions in Algorithm 9.1 are listed in Table 9.1.

#### 9.2.1.1 Extension of *DLB_tool* into *DLS4LB*

We extended the *DLB_tool* into the dynamic loop scheduling for load balancing library (*DLS4LB*) with more DLS techniques, namely: SS, FSC, TSS, WF, and AWF. Several

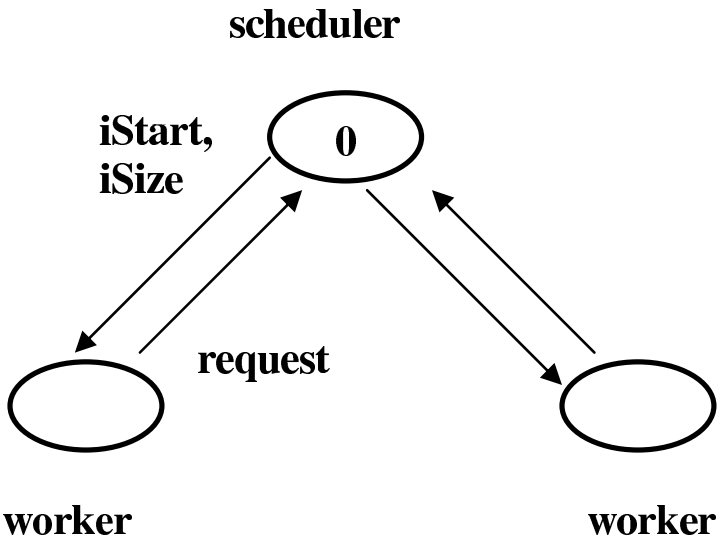



**Figure 9.3** **Master-worker scheduling model employed in the *DLB_tool*.** The master, rank 0, is serving work requests from workers by assigning them the start of the chunk and chunk size [CB07].

**Algorithm 9.1 Centralized MPI-Based DLS implementation**

```
#include <mpi.h>
#include "DLB_tool"
int main()
{
/* Application initialization                    */
...
DLS_setup(info, MPI_COMM)
DLS_Start_loop(info, P, N, DLS_method)
while (! DLS_Terminated(info)) do
    DLS_Start_chunk(info, loop_start, loop_end);
    for (i = loop_start; i < loop_end; i = i + 1) do
        /* Execute loop body */
        ...
    DLS_End_chunk(info)
DLS_End_loop(info)
...
}
```

modifications were required to achieve this extension. The chunk calculation functions of these DLS techniques were added to the code of the tool in C and FORTRAN. Additional data and inputs to `DLS_Setup` were added to support FSC and WF, such as the scheduling overhead $h$, the standard deviation of task execution times $\sigma$, and PEs weights.

**Table 9.1 Description of *DLB_tool* functions**

| Function name | Description |
|---|---|
| DLS_Setup | All: Initializes a dynamic loop scheduling environment. |
| DLS_Start_loop | Master: Specifies P, the number of PEs, N, the loop range, the DLS method and send the first chunks to workers. |
| DLS_Terminated | All: Returns true if all loop iterates have been executed. |
| DLS_Start_chunk | Master: Serve work requests and allocate new chunks.<br>Worker: wait for a new chunk or a terminate signal from the master. |
| DLS_End_chunk | Master: Collect performance data.<br>Worker: requests a new chunk of work, send performance data. |
| DLS_End_loop | All: synchronize after the loop completes. |

Given that time-stepping applications are the most common in scientific computing, AWF, which is designed for time-stepping applications, is added to the *DLS4LB*. Several changes were performed to support AWF. First, data allocation and deallocation are performed in `DLS_Setup` and `DLS_Finalize`, newly introduced functions, to be performed outside of the time-stepping loop to avoid allocating and deallocating with every time-step. Second, the adaptive weights of PEs are calculated in `DLS_Start_loop` instead of `DLS_Start_chunk` for AWF-B, AWF-C, AWF-D, and AWF-E. Also, `DLS_End_loop` collects and sends PEs performance data to the master, similar to `DLS_End_chunk` with other adaptive techniques. Finally, for better support for the time-stepping applications in AWF-B, AWF-C, AWF-D, and AWF-E, adaptive weights that are learned from previous time-steps are copied as the initial weights in the current time-step. Therefore, AWF-B, AWF-C, AWF-D, and AWF-E techniques do not need to start with a test chunk to calculate the weights as in non-time-stepping applications. Using weights learned in previous time-steps improves their load balancing and, consequently, the application performance.

### 9.2.2 Decentralized Coordination

We present a decentralized MPI-based implementation of the DLS techniques. We eliminate the master, which can be a performance bottleneck and a single point of failure. The current implementation leverages the MPI one-sided communication, such that processes calculate and allocate new chunks for themselves, without interrupting or requiring a master. A data owner process, instead of the master, holds the shared data (e.g., the current loop index of the loop and the scheduling step) between processes. The data owner process exposes these data and shares them with all other processes. All processes calculate and allocate chunks and update these shared variables using atomic op-

erations and locks, similar to the thread level implementation above (Section 9.1 and Figure 7.3(b)).

Algorithm 9.2 shows the one-sided approach to decentralize the DLS implementation. As MPI one-sided does not provide locks or critical sections, the `MPI_Get_accumulate` atomic operation is used to create a critical section to protect the shared data and synchronize access to it. We note that for simple DLS techniques, such as SS, FSC, and GSS, a lock or a critical section is not needed, and they can be implemented using only `MPI_Get_accumulate` atomic operation. However, to support more complex adaptive techniques, creating a lock was unavoidable. This approach was used to implement SS, FSC, GSS, FAC, WF, AWF-B, AWF-C, AWF-D, AWF-E, and AF using the decentralized coordination approach [MC18b].

A similar approach of employing the MPI one-sided passive target communication approach to decentralize the implementation of five DLS techniques was pro-

**Algorithm 9.2 Decentralized MPI-Based DLS implementation**

```
#include <mpi.h>
int main()
{
/* shared_data contains current loop index, scheduling
   step, PE adaptive weights, ..etc                    */
1  Allocate sem, shared_data
2  /* Application initialization*/
3  MPI_Win_create(sem)
4  MPI_Win_create(shared_data)
5  ...
6  while (True)) do
7      MPI_Win_lock(MPI_LOCK_EXCLUSIVE,sem)
8      MPI_Get_accumulate(&minus_one,sem,MPI_SUM)
9      if sem < 0 then
10         BLOCKED = True
11         continue
12     BLOCKED = FALSE
13     Calculate_chunk(N, P, shared_data)
14     MPI_Win_lock(MPI_LOCK_EXCLUSIVE,shared_data)
15     MPI_Put(shared_data, dataowner)
16     MPI_Win_unlock(shared_data)
17     for (i = loop_start; i < loop_end; i = i + 1) do
18         /* Execute loop body */
19         ...
20 ...
21 }
```

posed [EC19a]. Experiments therein confirm the superior performance of the decentralized approach to the centralized master-worker implementation. The distribution of the chunk calculation among all PEs in the decentralized approach not only improved the performance but also made it robust to the mapping of the master (or data owner) process on a slow or fast compute node [EC19a].

# 10

# Single-level Load Balancing of Scientific Applications

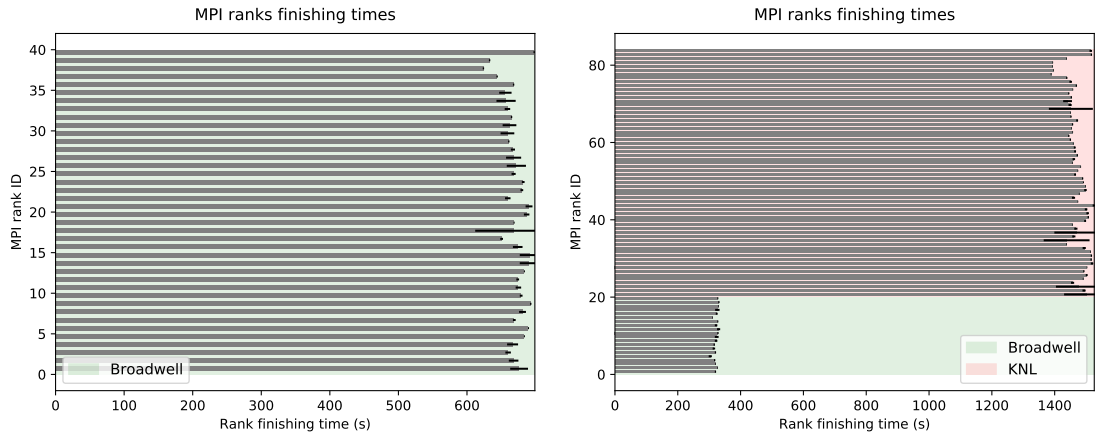
In the previous chapters, we have shown the most successful and well-known DLS techniques (Chapter 2), how to verify that their implementation adheres to their specification (Chapter 7), and how to implement them in OpenMP and MPI (Chapter 9). In this chapter, we show the advantage and the performance improvement of employing DLS techniques to balance the load of a real application from computer vision and Mandelbrot application as a scheduling benchmark. We study the use of nonadaptive and adaptive DLS techniques in balancing the execution of PSIA and Mandelbrot on homogeneous and heterogeneous HPC systems and analyze the performance of the applications.

## 10.1 Load Imbalance in PSIA and Mandelbrot

### 10.1.1 Load Imbalance in PSIA

The PSIA [EFM+16] is an application from computer vision. It converts a 3D object into a set of 2D descriptors that represent that object. PSIA is used in face detection [CK13], object recognition [JH99], 3D map registration [MH13], 3D data retrieval [ADADB+ne]. The pseudocode for PSIA is presented above in Algorithm 8.1, and Figure 8.2 shows the process of generating spin-images.

PSIA is parallelized using MPI, where each MPI rank generates a certain amount of spin-images using the *DLS4LB*. The input data are broadcasted (replicated) on all MPI processes. Therefore, the master only assigns work by sending workers loop indices (start and end), and no data are moved between ranks. Workers generate the spin-images assigned to them and send the generated images only after all work is completed (one gather operation at the end of the execution).




(a) MPI ranks execution time on two homogeneous nodes

(b) MPI ranks execution time on two heterogeneous nodes

**Figure 10.1 Impact of load imbalance on the performance of PSIA on homogeneous and heterogeneous HPC systems.** Processes finishing times executing PSIA to generate 400, 000 spin-images on two homogeneous (20 + 20 Broadwell cores, Figure 10.1(a)) and heterogeneous (20 Broadwell + 64 KNL cores, Figure 10.1(b)) nodes of miniHPC. The gray horizontal bars represent the median process execution time over 20 repetitions, and the black bars represent their standard deviation. The variation in processes finishing times indicates load imbalance of PSIA with STATIC straightforward scheduling.

The source of load imbalance in PSIA is the conditional execution of certain computations (see Algorithm 8.1) according to the input data. Also, the difference in the computing speed of the PEs contributes to the load imbalance. Figure 10.1 shows the load imbalance in PSIA with STATIC in generating 400, 000 spin-images using two nodes of miniHPC (see Section 8.2.1). The experiment was performed on homogeneous cores, i.e., using two Broadwell nodes and 20 cores per each node and on heterogeneous cores as well, using one Broadwell node (20 cores) and one KNL node (64 cores) to investigate the load imbalance on homogeneous and heterogeneous systems.

On homogeneous cores, PSIA execution does not incur high load imbalance, as can be seen in Figure 10.1(a). The mean/max of ranks finishing times is 0.960 and their c.o.v. is 0.023 (recall load balance metrics in Section 2.1). However, the execution of PSIA on heterogeneous cores incurs high load imbalance due to the significant performance difference between Broadwell and KNL cores, as can be seen in Figure 10.1(b). The mean/max of ranks finishing times is 0.782, and their c.o.v. is 0.409. The significantly high value of c.o.v. and the low value of mean/max of ranks finishing times on heterogeneous cores compared to homogeneous ones shows the impact of system heterogeneity on the application load balance.

#### 10.1.1.1 Static load balancing of PSIA

Empirical static division of work has been used previously to balance the load of PSIA executing on an Intel CPU and Intel Knights Corner (KNC) co-processor accelerator [EFM+16]. The work was divided unequally between the two PEs in such a way that it achieves a balanced execution. However, the approach therein is very problem-system specific and needs many trials and experimentations to reach the golden ratio of the division of work that achieves the load balance.

### 10.1.2 Load Imbalance in Mandelbrot

Mandelbrot application computes the Mandelbrot set [Man80] and generates its image. The application is parallelized at the MPI level with the *DLS4LB* such that the calculation of the value at every single pixel of a 2D image is a loop iteration that is performed in parallel. The application computes the function

$$f_c(z) = z^4 + c \tag{10.1}$$

instead of

$$f_c(z) = z^2 + c \tag{10.2}$$

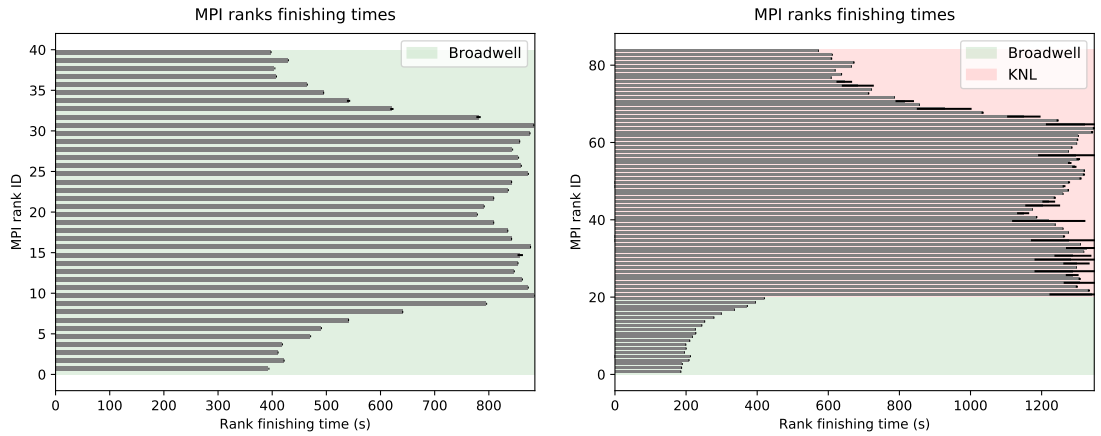
to increase the number of computations per task (see Algorithm 8.2). The number of calculations per task is irregular, which represents the source of load imbalance in Mandelbrot (see Section 8.1). The size of the generated image is $512 \times 512$ pixels resulting in $2^{18}$ parallel tasks. Mandelbrot is often used as a scheduling benchmark due to its application-induced high load imbalance.

Unlike PSIA, Mandelbrot execution incurs high load imbalance, as can be seen in Figure 10.2(a) on homogeneous cores. The mean/max of ranks finishing times is 0.785, and their c.o.v. is 0.273. On heterogeneous cores, the load imbalance increases due to the system heterogeneity, as seen in Figure 10.2(b). The mean/max of ranks finishing times is 0.667, and their c.o.v. is 0.477. The results show that system heterogeneity increases the load imbalance incurred by Mandelbrot execution.

## 10.2 Balancing Applications on Homogeneous and Heterogeneous HPC Systems

### 10.2.1 Design of Experiments

Several experiments are designed to investigate the load balancing of PSIA and Mandelbrot with DLS on homogeneous and heterogeneous HPC systems. Table 10.1 lists an




(a) MPI ranks execution time on two homogeneous nodes

(b) MPI ranks execution time on two heterogeneous nodes

**Figure 10.2 Impact of load imbalance on the performance of Mandelbrot on homogeneous and heterogeneous HPC systems.** Processes finishing times executing Mandelbrot to generate $512 \times 512$ pixels Mandelbrot image on two homogeneous (20 + 20 Broadwell cores, Figure 10.2(a)) and heterogeneous (20 Broadwell + 64 KNL cores, Figure 10.2(b)) nodes of miniHPC. The gray horizontal bars represent the median process execution time over 20 repetitions, and the black bars represent their standard deviation. The variation in processes finishing times indicates load imbalance of Mandelbrot with STATIC straightforward scheduling.

overview of the details of these experiments. Applications performance is tested without DLS, i.e., with STATIC, and with mFSC, GSS, FAC, and AF DLS techniques. Two configurations of the miniHPC system are used to represent homogeneous and heterogeneous systems. The first configuration is using eight Broadwell nodes to create a homogeneous cluster. In the second configuration, Broadwell and KNL partitions of the miniHPC system (see Section 8.2.1) are combined to create a heterogeneous HPC system. The performance of both applications is examined in weak and strong scaling experiments.

In *weak scaling* experiments, the number of generated spin-images per node is fixed to 50, 000 per node for PSIA. The number of nodes is increased up to eight (homogeneous or heterogeneous) nodes. In *strong scaling* experiments, the problem size (number of generated spin-images) is fixed to 400, 000 while increasing the number of nodes used to generate the spin-images. For Mandelbrot, the number of tasks quadratically increases as the number of tasks is the size of a 2D image, i.e., length $\times$ width. Therefore, in weak scaling experiments, the number of tasks is kept proportional to the number of nodes, using $256^2, 362^2, 443^2$, and $512^2$ tasks for $2, 4, 6$, and 8 nodes, respectively. In strong scaling experiments, the number of tasks is fixed to $512^2$ tasks while increasing the system size from 2 to 8 (homogeneous or heterogeneous) nodes.

**Table 10.1 Details of PSIA and Mandelbrot load balancing experiments**

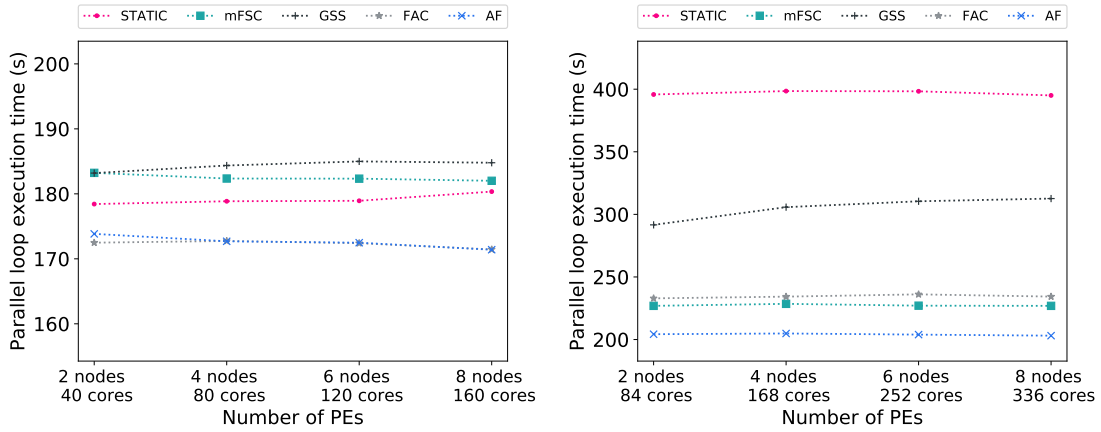
| Factors | Values | Properties |
|---|---|---|
| **Applications** | PSIA | $N = 4000,000$ tasks |
| | Mandelbrot | $N = 262,144$ tasks |
| **Loop Scheduling** | STATIC | Static |
| | mFSC<br>GSS<br>FAC | Nonadaptive dynamic |
| | AF | Adaptive dynamic |
| **Computing system** | miniHPC | 8 Broadwell nodes, $8 \times 20 = 160$ homogeneous cores<br>4 Broadwell +4 KNL nodes = $4 \times 20 + 4 \times 64 = 336$ heterogeneous cores |
| **Experimentation** | Weak scaling | 2, 4, 6, 8 homogeneous and heterogeneous<br>PSIA: $N = 100, 200, 300, 400 \times 10^3$ tasks<br>Mandelbrot: $N = 256^2, 362^2, 443^2, 512^2$ tasks |
| | Strong scaling | 2, 4, 6, 8 homogeneous and heterogeneous<br>PSIA: $N = 400 \times 10^3$ tasks<br>Mandelbrot: $N = 512^2$ tasks |

## 10.2.2 Results and Discussion

### 10.2.2.1 Weak scalability

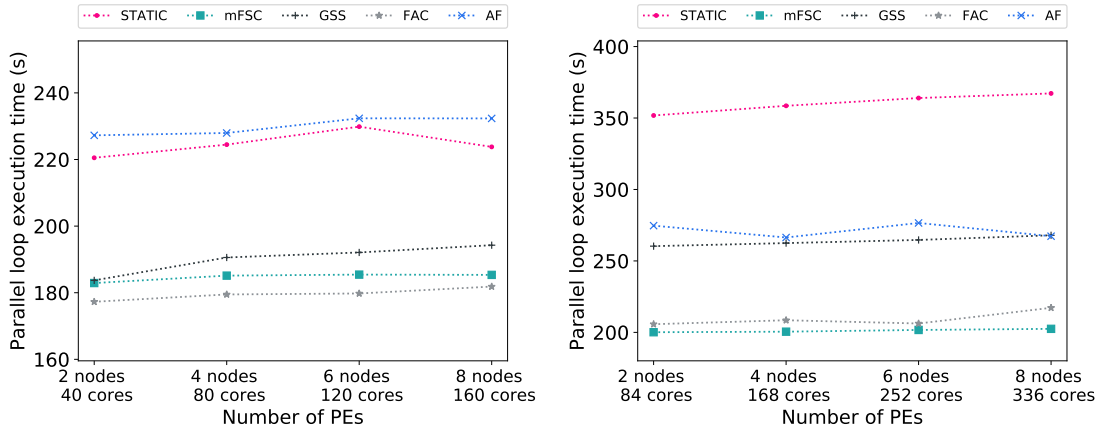
The results of the weak scalability experiments of PSIA are presented in Figure 10.3. A straight horizontal line represents perfect scalability. One can notice that FAC and AF improved the performance of PSIA on homogeneous cores by almost 5% (see Figure 10.3(a)). While the execution time of PSIA increases with STATIC while scaling from two to eight nodes, it decreases with FAC and AF by about 3%. The better scaling profile of PSIA with FAC is attributed to the benefit of load balancing while scaling, which is confirmed by the degraded scalability of PSIA with STATIC. Due to the mild load imbalance of PSIA on homogeneous systems, mFSC and GSS degrade PSIA's performance on homogeneous cores due to their unjustified overhead and low load imbalanced incurred during the execution.

Unlike the performance on homogeneous cores, all DLS techniques significantly improved PSIA's performance on heterogeneous cores (see Figure 10.3(b)). The AF technique outperformed all other techniques and improved the performance by almost 100% while maintaining perfect scalability. mFSC and FAC also significantly improved PSIA's performance and scaled perfectly. However, PSIA's performance with GSS degrades while scaling. GSS assigns large chunks of tasks at the beginning of the execution, which can cause load imbalance.




(a) Weak scalability on homogeneous cores (b) Weak scalability on heterogeneous cores

**Figure 10.3 The *weak* scaling performance results of PSIA with DLS techniques on homogeneous and heterogeneous cores.**




(a) Weak scalability on homogeneous cores (b) Weak scalability on heterogeneous cores

**Figure 10.4 The *weak* scaling performance results of Mandelbrot with DLS techniques on homogeneous and heterogeneous cores.**

The results of the weak scalability experiments of Mandelbrot are presented in Figure 10.4. Unlike PSIA, FAC outperforms all other DLS techniques on homogeneous cores and improves the performance by up to 20%. Surprisingly, AF degrades Mandelbrot's performance. This is due to the small granularity of Mandelbrot tasks and their high variability, which hinder the learning of the adaptive weights of AF.

Similar to the performance on homogeneous cores, mFSC and FAC significantly improve Mandelbrot's performance on heterogeneous cores (see Figure 10.4(b)). The mFSC technique outperformed all other techniques and improved the performance by almost 75%, maintaining perfect scalability. GSS and AF also significantly improved Mandelbrot's performance and scaled almost perfectly. However, FAC improved the perfor-

mance significantly; it incurs a performance degradation by 10% while scaling from 6 to 8 nodes.

#### 10.2.2.2 Strong scalability

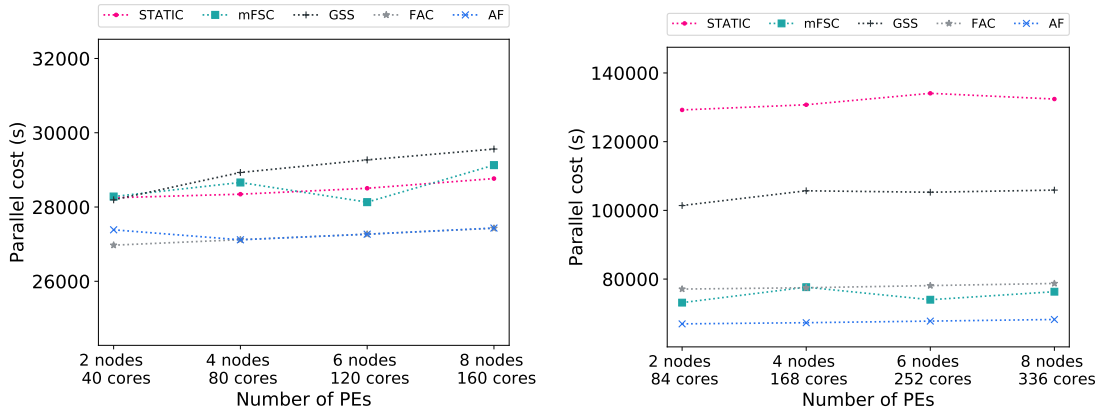
Figure 10.5 shows the results of the strong scaling of PSIA on the homogeneous and heterogeneous cores of the miniHPC. The parallel cost is calculated as the number of PEs multiplied by the parallel execution time. Parallel cost is reported instead of the parallel execution time for this experiment, as it reflects the benefits of using additional computing resources versus the time needed to execute the application. A perfect strong scalability profile of a program corresponds to an almost constant parallel cost for any number of computing resources.

Figure 10.5(a) shows the parallel cost of PSIA on homogeneous cores. The results confirm the superior performance of FAC and AF with PSIA over all other DLS techniques, and the poor performance of the GSS technique. The parallel cost of PSIA significantly increases with STATIC, mFSC, and GSS, which reflects the poor strong scalability of PSIA with such DLS techniques on homogeneous cores. Using AF with PSIA improved the application performance while scaling by 3% compared to STATIC, and by 6% compared to GSS.

Strong scalability results of PSIA on heterogeneous cores of miniHPC are shown in Figure 10.5(b). Similar to the results on homogeneous cores, AF outperforms all other techniques and scales almost perfectly. Unlike results on homogeneous cores, mFSC outperformed FAC on heterogeneous cores. FAC assumes a homogeneous computing system. Therefore, it fails to balance the load on heterogeneous cores. The parallel cost of PSIA increased while scaling with STATIC, mFSC, and GSS, which indicates their suboptimal scalability. Comparing to performance with STATIC, AF improved PSIA's performance by 85% and scaled perfectly on heterogeneous cores.

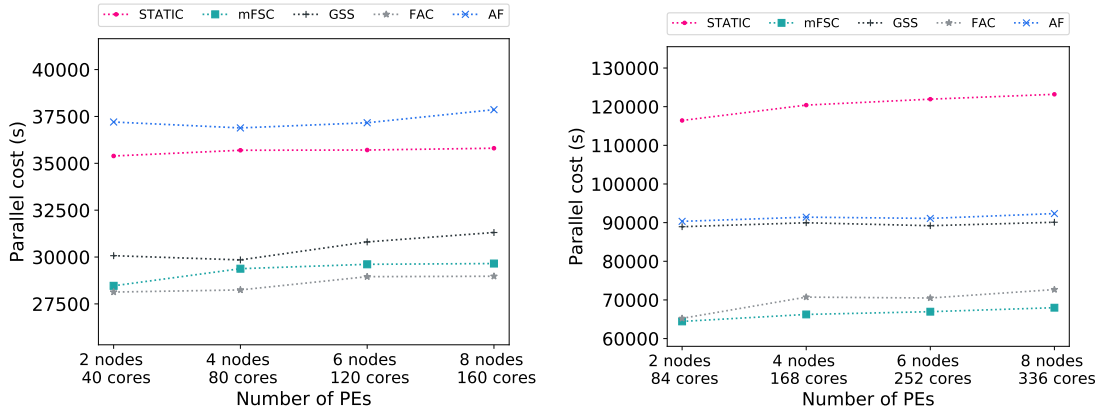
Strong scalability results of Mandelbrot on homogeneous cores of miniHPC are presented in Figure 10.6(a). Unlike PSIA, AF delivers the poorest performance while FAC achieves the best performance, similar to PSIA. FAC improves Mandelbrot's performance by 36% compared to STATIC. However, FAC does not scale perfectly, and the parallel cost of Mandelbrot is increased by almost 3% while scaling. mFSC achieves a similar performance and scalability of FAC, while GSS delivers poorer performance in terms of parallel cost and scalability.

The performance results of the strong scalability experiments of Mandelbrot on heterogeneous cores of miniHPC are shown in Figure 10.6(b). Unlike results on homogeneous cores, mFSC outperformed FAC on heterogeneous cores and achieved the best performance. Performance with GSS and AF is almost similar. Both techniques im-




(a) Strong scalability on homogeneous cores (b) Strong scalability on heterogeneous cores

**Figure 10.5 The *strong* scaling performance results of PSIA with DLS techniques on homogeneous and heterogeneous cores.**



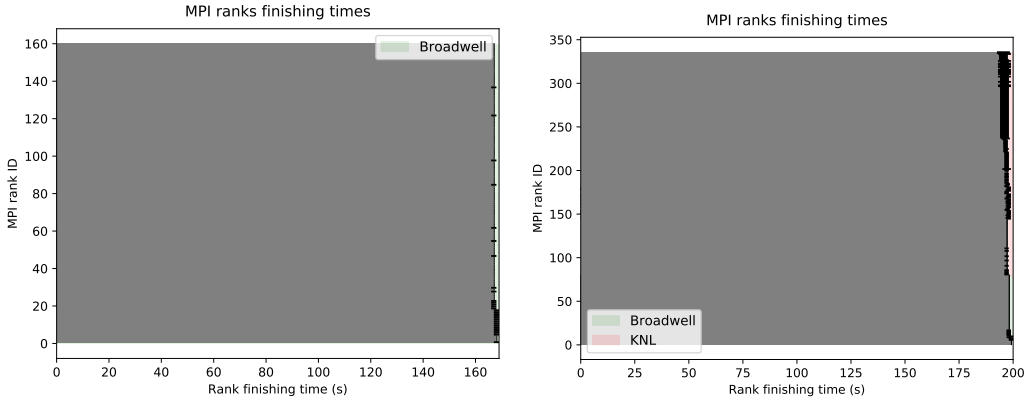

(a) Strong scalability on homogeneous cores (b) Strong scalability on heterogeneous cores

**Figure 10.6 The *strong* scaling performance results of Mandelbrot with DLS techniques on homogeneous and heterogeneous cores.**

proved the performance by 35% compared to STATIC and scaled perfectly. The parallel cost of Mandelbrot increased by almost 6% while scaling with mFSC and FAC despite the improved performance. mFSC improved Mandelbrot's performance by almost 92% compared to performance with STATIC.

### 10.2.3 Discussion

In this subsection, we revisit the load balance metrics calculated for PSIA and Mandelbrot execution on homogeneous and heterogeneous cores in Section 10.1. We measure processes execution times with the most efficient DLS technique in the weak and strong scalability experiments and calculate the load balance metrics for these executions to




(a) Load balancing with AF, homogeneous cores

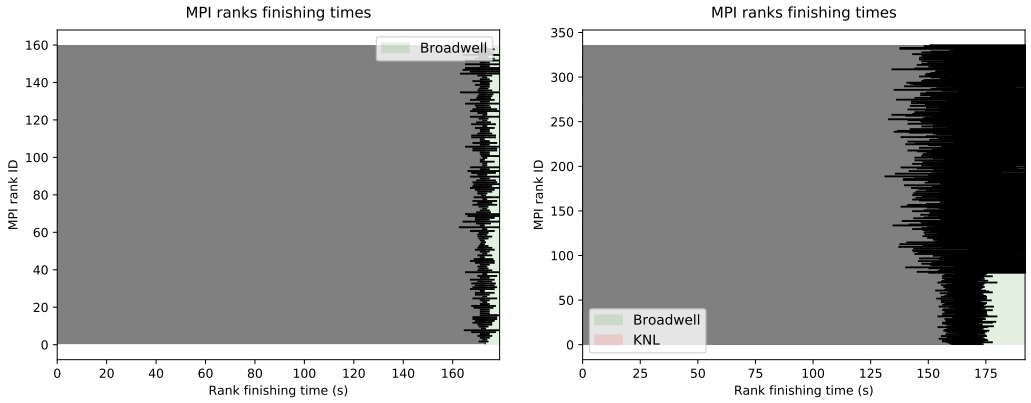
(b) Load balancing with AF, heterogenous cores

**Figure 10.7 Impact of dynamic load balancing on the performance of PSIA.** Processes finishing times executing PSIA to generate 400, 000 spin-images on eight homogeneous (160 Broadwell cores, Figure 10.7(a)) and heterogeneous (80 Broadwell + 256 KNL cores, 10.7(b)) nodes of miniHPC. The gray horizontal bars represent the median process execution time over 20 repetitions, and the black bars represent their standard deviation. Load balancing with AF improved PSIA's performance and scalability.

show the improvement in load balancing, as well as performance and scalability of PSIA and Mandelbrot with DLS techniques on homogeneous and heterogeneous cores.

Figure 10.7 confirms that the improved performance and scalability of PSIA with AF is due to load balancing. All processes almost finish at the same time on homogeneous and heterogeneous cores due to the load balancing of AF. The load imbalance metrics for the execution of PSIA on homogeneous cores are mean/max = 0.995 and c.o.v. = 0.002. For heterogeneous cores, the mean/max and c.o.v. are 0.989 and 0.006, respectively. Comparing the load balance metrics with AF with those obtained with STATIC in Section 10.1, one can see the significant improvement of load balance with AF.

Similarly, for Mandelbrot in Figure 10.8, the results confirm that the improved performance and scalability of Mandelbrot due to load balancing. The load balance metrics for the execution of Mandelbrot on homogeneous cores with FAC (Figure 10.8(a)) are mean/max = 0.956 and c.o.v. = 0.009. For heterogeneous cores, the mean/max and c.o.v. are 0.872 and 0.061, respectively. Comparing the load balance metrics obtained with FAC and mFSC with those obtained with STATIC in Section 10.1, one can see the significant improvement of load balance both, on homogeneous and heterogeneous cores, respectively. *The results of strong and weak scalability experiments show and confirm that load imbalance degrades applications performance and becomes more significant at large scale.*




(a) Load balancing with FAC, homogeneous cores

(b) Load balancing with mFSC, heterogeneous cores

**Figure 10.8 Impact of dynamic load balancing on the performance of Mandelbrot.** Processes finishing times executing Mandelbrot to generate 512×512 pixels Mandelbrot image on eight homogeneous (160 Broadwell cores, Figure 10.8(a)) and heterogeneous (80 Broadwell + 256 KNL cores, Figure 10.8(b)) nodes of miniHPC. The gray horizontal bars represent the median process execution time over 20 repetitions, and the black bars represent their standard deviation. Load balancing with FAC and mFSC improved Mandelbrot's performance and scalability on homogeneous and heterogeneous cores, respectively. The high standard deviation of processes finishing times (black bars) in Figure 10.8(b)) is due to the dynamic scheduling, i.e., ranks do not take the same chunks in the 20 repetitions of the execution and due to the high variability of Mandelbrot task sizes.

# 11

# Two-level Load Balancing of Scientific Applications

HPC systems are growing vertically in the number of PEs (cores) available in a shared memory node, and horizontally in the number of nodes per system. Consequently, applications use a hybrid process and thread parallelism to exploit multiple hardware parallelism levels. As the number of cores per node and number of nodes per system increase, performance degradation due to load imbalance becomes more significant [EMC17a] (see Chapter 10).

The analysis of delays caused by wait states and their propagation in parallel MPI+OpenMP applications identified two-level load imbalance as a major challenge for Peta- and Exascale systems [BGW+10; BGA+16]. Redistribution of excess workload that is the root-cause of wait states solves this problem. In this chapter, we investigate the use of DLS techniques to load balance scientific applications at the two-levels dynamically, namely the process level and the thread level. We employ the *eLaPeSD* and *DLS4LB* developed in Chapter 9 to achieve two-level dynamic load balancing in MPI+OpenMP scientific applications (see Figure 11.1). We investigate the influence of dynamic load balancing using DLS on the process level and the thread level and the interplay between the load balancing at the two levels.

## 11.1 Two-level Dynamic Load Balancing

Two-level load imbalance may manifest in process+thread (MPI+OpenMP) parallel scientific applications. Figure 11.2 illustrates the *two-level dynamic load balancing via self-scheduling* approach. At the process level, processes are self-scheduled chunks of tasks whenever they are free and request work. The work assigned to a process is parallelized and distributed among several threads (8 threads per process in Figure 11.2) using

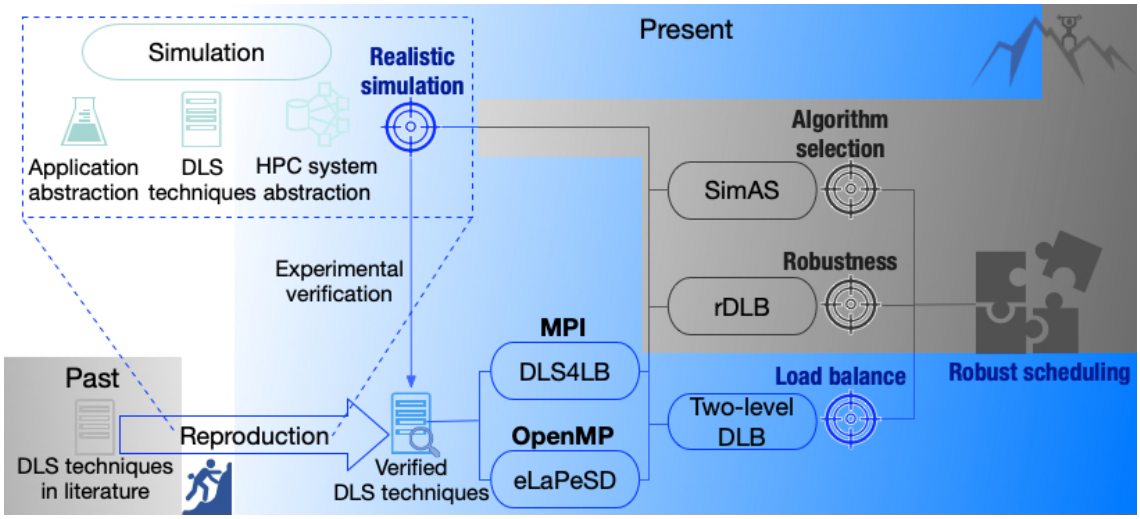



**Figure 11.1 Illustration of the focus and the progress towards robust scheduling (in colors) as part of the overall approach (in grayscale).** Verified DLS implementations are used for the dynamic load balancing of scientific applications at the thread level and process level.

self-scheduling for load balancing. Employing DLS at only one level (either process or thread level) achieves load balancing at this level and improves the application performance, as shown in the middle subplots in Figure 11.2. However, the best application performance can only be achieved by employing dynamic load balancing at both levels, as shown in Figure 11.2 (the rightmost subplot).

Based on the selected DLS technique at the process level and the selected scheduling technique at the thread level, along with the application and the computing system properties, different degrees of load balancing and performance improvement are achieved. The two-level dynamic load balancing via the self-scheduling approach depicted in Figure 11.2 is generic and can be used with any scientific application with independent tasks.

### 11.1.1 Implementation

Due to the hybrid nature of current HPC systems, distributed memory across compute nodes and shared memory within single nodes, hybrid parallelization of scientific applications at process level and thread level using MPI+OpenMP is the most common and successful approach [JJM+11; RHJ09; SB01]. Here we consider the implementations of DLS techniques described in Chapter 9 for the two-level dynamic load balancing of MPI+OpenMP applications. To balance the load at the thread level, an extended version of GNU OpenMP runtime library *eLaPeSD* [CIB18] is used (see Section 9.1). *eLaPeSD* supports seven OpenMP scheduling techniques that can be selected by exporting the name of the scheduling technique to the OpenMP environment variable OMP_SCHED–

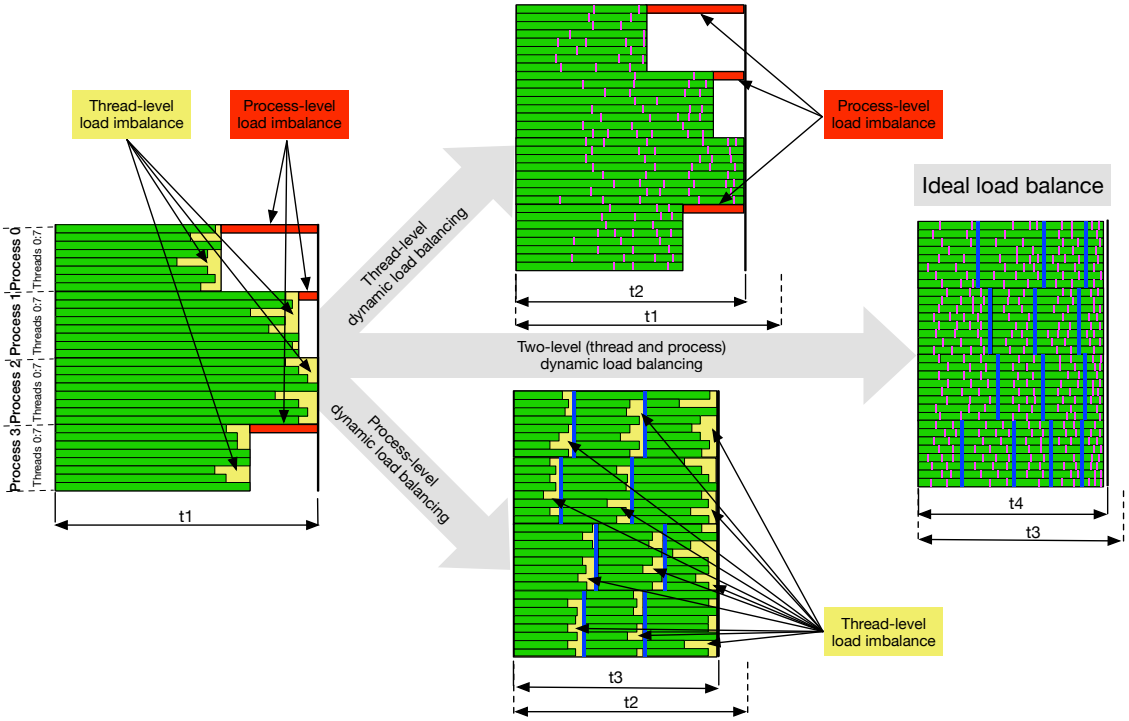



**Figure 11.2 Conceptual illustration of employing two-level dynamic load balancing at thread level and process level.** At the process level, chunks of work (tasks or loop iterations) are self-scheduled to free and requesting processes in multiple rounds. At the thread level, threads are self-scheduled chunks of tasks (assigned to their respective process) whenever they become free. Load balancing at a certain level improves application performance and balances the load at that level. However, two-level dynamic load balancing improves application performance and achieves a balanced load execution.

ULE. *eLaPeSD* supports static (STATIC), dynamic (SS), guided (GSS), FSC, TSS, FAC, and RAND scheduling techniques. To enable the application to read and use scheduling algorithms defined in the OpenMP runtime library, the `schedule(runtime)` needs to be added to the OpenMP parallelization of a `for` loop (in C) or a `do` loop (in FORTRAN), as shown in Algorithm 11.1 Line 15 (in magenta font color). The path to *eLaPeSD* library needs to be added to the dynamic library path variable `LD_LIBRARY_PATH` to call our extended OpenMP runtime library instead of the standard one. To use the FSC scheduling technique in *eLaPeSD*, two additional environment variables, namely: `SIGMA` and `FSCH`, are required to specify the standard deviation of the loop iterations execution times, $\sigma$, and the scheduling overhead, $h$, needed by the FSC to calculate the chunk size.

To balance the load at the process level, *DLS4LB* is used to distribute the application tasks to MPI ranks dynamically in multiple rounds via self-scheduling. The *DLS4LB* supports 14 loop scheduling techniques, ranging from fully static to fully dynamic, nonadap-

tive and adaptive, namely: STATIC, SS, FSC, mFSC, TSS, GSS, FAC, WF, AWF, AWF-B, AWF-C, AWF-D, AWF-E, and AF (see Section 9.2). *DLS4LB* is implemented in C and FORTRAN to be compatible with the two most common languages used in HPC applications. Calls to the *DLS4LB* need to be inserted before and after the main calculations (loop body or independent tasks) such that they are self-scheduled using DLS techniques as shown by lines in blue font color in Algorithm 11.1. Function calls for setting up and finalizing the *DLS4LB*, such as data allocation, deallocation, and selecting the DLS technique, needs to be added before and after the time-stepping loop (Algorithm 11.1, Lines 2 – 9), respectively.

**Algorithm 11.1 Two-level (MPI+OpenMP) Dynamic Load Balancing via Self-scheduling**

```
   #include <mpi.h>
   #include <omp.h>
   #include "DLS4LB.h"
   int main()
   {
   /* Application initialization                                    */
1  ...
2  DLS4LB_setup(info, MPI_COMM
   /* Time-stepping loop                                            */
3  for (l = t_init; l < t_final; l = l + 1) do
4  |  ...
5  |  DLS4LB_Start_loop(info, P, N, DLS_method)
6  |  Main_calculations()
7  |  DLS4LB_End_loop(info)
8  |  ...
9  DLS4LB_Finalize(info)
10 } /* End main                                                    */

11 void Main_calculations()
12 {
13 while (! DLS4LB_Terminated(info)) do
14 |  DLS4LB_Start_chunk(info, loop_start, loop_end)
15 |  #pragma omp parallel for schedule(runtime)
16 |  for (i = loop_start; i < loop_end; i = i + 1) do
   |  |  /* Execute loop body                                      */
17 |  |  ...
18 |  DLS4LB_End_chunk(info)
19 } /* end main calculations                                      */
```

### 11.1.2 Execution

Each MPI rank (process) is initially assigned a chunk of work by the master (Algorithm 11.1, Line 5). MPI ranks send a work request to the master rank whenever they become free (Algorithm 11.1, Line 18) using the *DLS4LB*. In response, the master rank assigns a chunk of tasks to the requesting MPI rank (Algorithm 11.1, Line 14). The size of the assigned chunk is determined by the employed DLS technique (specified in Algorithm 11.1, Line 5). *This allocated chunk at the MPI level is subsequently distributed to OpenMP threads for further scheduling and execution at the thread level.* Therefore, threads are assigned chunks (or sub-chunks of the chunk allocated at the MPI level) of tasks whenever they become free using *eLaPeSD*. Threads are assigned work until they complete the execution of the chunk allocated to their respective MPI rank. The process repeats until all tasks ($N$ tasks) complete (condition in Algorithm 11.1, Line 13 becomes false) and `Main_calculations` of the current time-step is completed.

### 11.1.3 Remarks

We note that an application needs to be a time-stepping application to use the AWF technique. Otherwise, an application may use all other DLS techniques available in the *DLS4LB* library. Additionally, the current implementation of the *DLS4LB* does not communicate the data required to compute the assigned chunk of loop iterations. It is left to the programmer to ensure that the data are available for computation and are correctly communicated or replicated on all MPI ranks. In the experiments included below, application data is either replicated on all ranks, or no data are required for the scheduled computations (see Section 11.2.1.1). Calls to the OpenMP runtime library, such as *eLaPeSD* or *DLS4LB*, incurs overhead, compared to the static scheduling. However, this overhead is justified and absorbed by the performance gain via dynamic load balancing (see Section 11.2.2).

## 11.2 Experimental Evaluation and Discussion

### 11.2.1 Design of Experiments

Three applications and 66 combinations of DLS techniques at the process level and thread level are experimented herein to investigate and analyze the performance of scientific applications with two-level dynamic load balancing. Table 11.1 summarizes the details of the experiments performed herein.

**Table 11.1 Details used in the design of two-level dynamic load balancing experiments.**

| Factors | Values | Properties |
|---|---|---|
| **Applications** | Mandelbrot (Mathematics) | $N = 0.6 \times 10^6$ tasks |
| | PSIA (Computer vision) | $N = 0.8 \times 10^6$ tasks |
| | SPHYNX (Astrophysics) | Test-case (1): Stellar collision<br>$N = 10.4 \times 10^6$ tasks<br>Time-step: 6900<br>Test-case (2): Evrard collapse<br>$N = 1 \times 10^6$ tasks<br>Time-step: 100, 500, 1000, 1700,<br>2000, 2300, 2500, 2800<br>full simulation[0 : 3000] |
| **Dynamic load balancing** | | |
| Thread-level self-scheduling | STATIC | Static: used as a baseline in this level |
| | SS, GSS, FSC*, TSS, FAC, RAND | Dynamic and nonadaptive |
| Process-level self-scheduling | NODLB | Static: used as a baseline at this level |
| | SS*, FSC*, mFSC, GSS, TSS, FAC, WF* | Dynamic and nonadaptive |
| | AWF, AWF-B, -C, -D, -E, AF | Dynamic and adaptive |
| **Computing systems** | miniHPC | 20 Dual socket Intel Broadwell nodes |
| **Experimentation** | Native | (1) Test different combinations of DLS at the process level and thread level<br>(2) Use the identified best two-level combination to improve the performance |

* DLS techniques implemented in the *eLaPeSD* or the *eDLS4LB* libraries but not used in this work due to being unsuitable (SS, WF) or requiring profiling (FSC).

#### 11.2.1.1 Applications

Three applications are considered in this work: Mandelbrot, PSIA, and SPHYNX, which represent applications from three different domains: mathematics, computer vision, and astrophysics, respectively.

`Mandelbrot` is a computationally-intensive application which computes the Mandelbrot set [Man80] and generates its image. The pseudocode (Algorithm 8.2) and details about Mandelbrot are described above (see Section 8.1). Mandelbrot is often used to evaluate the performance of dynamic scheduling techniques due to the high variation between its task execution times. The calculation is focused on the center of the image, where the computation is highly intensive and variable.

*Parallelization.* The application is parallelized such that the calculation of the value at every single pixel of a 2D image is a task that is performed in parallel. Mandelbrot code is

implemented in C, and MPI+OpenMP are used to parallelize the application. Mandelbrot is compiled with GNU GCC version 6.3 and OpenMPI 2.0.2 with *O*3 optimization level.
*Data distribution.* No data are required for the calculation of the Mandelbrot tasks, i.e., the value of a single pixel. Only the location of the pixel is required, which represents the task ID assigned by the DLS techniques.

PSIA is an application from the computer vision domain, namely the parallel spin-image algorithm (PSIA) [EFM+16]. PSIA converts a 3D object into a set of 2D descriptors (spin-images). The pseudocode of the PSIA (Algorithm 8.1) is described as above (see Section 8.1). The amount of computation required to generate the spin-images is data-dependent and is not identical over all the spin-images generated from the same object, resulting in load imbalance.
*Parallelization.* The computation of each spin-image is considered a task. PSIA is implemented in C, parallelized with MPI+OpenMP, and compiled with GNU GCC version 6.3 and OpenMPI 2.0.2 with *O*3 compiler optimizations.
*Data distribution.* The input 3D object is read at the beginning by MPI rank 0 and is broadcasted to all other ranks. During loop execution, only spin-image ID needs to be communicated to other ranks to generate the required image, i.e., task ID (loop index). Therefore, no data communication is required. After computation completes, all data are aggregated back to MPI rank 0 for output.

SPHYNX is a state-of-the-art production smoothed particle hydrodynamics (SPH) code[1] [CGSF17]. SPHYNX is the first to include an integral approach to calculate derivatives (IAD) and a dynamically adaptive interpolation *sinc* kernel, being both elements of great importance in overcoming long-lasting problems of the SPH technique related to the development of subsonic hydrodynamical instabilities [CGR08; GCE12]. It is also one of the few hydrodynamics codes in the literature that can simulate both Type Ia and core-collapse Supernovas, including nuclear reactions, neutrino transport, and general relativity correction terms. The main structure of SPHYNX is described in Algorithm 11.2. After the initialization of all variables, the code proceeds to perform several time iterations (or time-steps), evolving the system. Each time iteration performs a series of computational steps (enumerated in Algorithm 11.2), beginning with the creation of a tree structure used to locate neighboring particles and to calculate self-gravity. Then, a list of neighbors for each particle is found, this step being central for the SPH technique. Once this list is known, the SPH interpolations can be performed to find the density distribution, the IAD terms, the thermodynamical properties of the system, and the rates of change of velocities and internal energy. Finally, the particle positions, velocities, and internal energy are updated, a new time-step is found, and a new time iteration begins.

[1] Available at http://astro.physik.unibas.ch/sphynx

**Algorithm 11.2 SPHYNX Computational Workflow**

**for** $l \leftarrow t_{init}$ **to** $t_{final}$ **do**

1. Build tree
2. Find neighbors
   2.1 Collective communication (number of neighbors)
3. Density & grad-h calculations
   3.1 Collective communication (density & grad-h)
4. IAD calculations
   4.1 Collective communication (IAD terms)
5. EOS & $\nabla\mathbf{v}$ calculations
   5.1 Collective communication ($\nabla \cdot \mathbf{v}$ & $\nabla \times \mathbf{v}$)
6. Momentum & energy calculations
   6.1 Collective communication ($\nabla P$ & $du/dt$)
7. Gravity calculations
   7.1 Collective communication (gravitational force and potential)
8. Update velocities, position, and energy
9. Time-step evaluation
10. Verification via conservation laws

The performance of SPHYNX is studied for two simulation test-cases. The *stellar collision* test simulates the head-on impact of two Sun-like stars. This simulation has two independent gravitating bodies and, therefore, the particle distribution is highly asymmetric. Second, the *Evrard collapse* is a common test used to examine the coupling between hydrodynamics and self-gravity in astrophysical codes. It simulates the collapse of an unstable cloud of gas and the formation of the subsequent shockwave. The two test-cases offer a wide range of problem sizes, defined in the number of particles in the system and different particle distributions that represent different load balancing challenges.

*Parallelization.* SPHYNX is a time-stepping application, written in FORTRAN F90 and parallelized using MPI and OpenMP. SPHYNX was also compiled with GNU compilers version 6.3 with O3 optimization, and OpenMPI 2.0.2 is also used similar to Mandelbrot and PSIA.

*Data distribution.* The data of the SPH particles and neighbors per particle are replicated on all MPI ranks. The data are kept updated after each computational step by collective communications, as shown in Algorithm 11.2.

#### 11.2.1.2 Load imbalance in the considered applications

Figure 11.3 shows the load imbalance in the 3 applications considered with no load balancing at the thread and process level (i.e., default STATIC) executing on miniHPC. This load imbalance manifests as overhead (i.e., waiting time) as depicted in Figure 11.3 at the

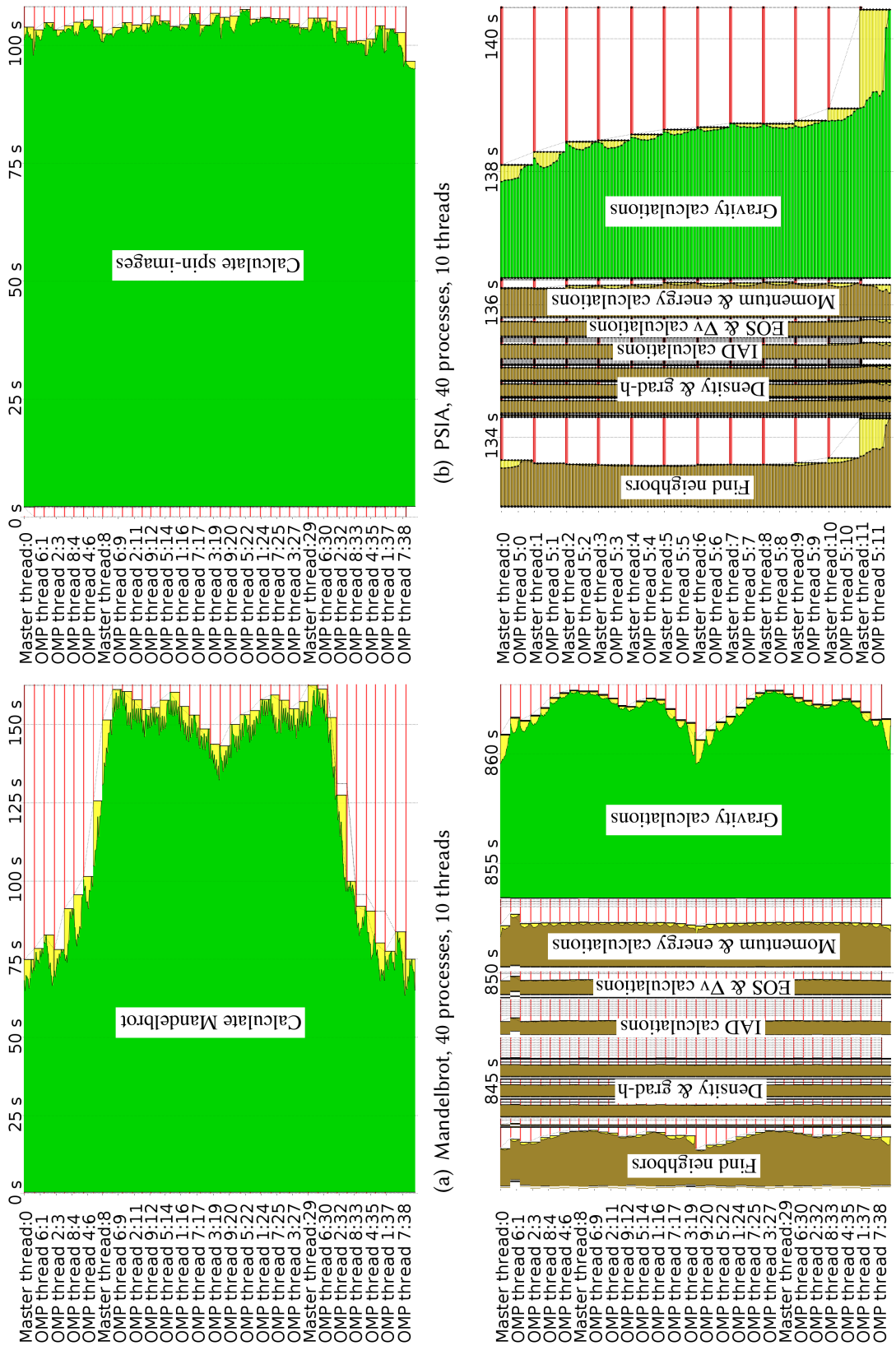
thread-level (yellow regions) and at the process-level (red regions). Calculating gravity in SPHYNX is the most time-consuming computational step and also the most load imbalanced. Therefore, herein we focus on improving the gravity calculation step of SPHYNX. To obtain representative performance measurements, the two-level load balancing is tested in the middle of the simulation time (time-step 6900) for the *stellar collision* test-case. For the Evrard collapse test-case, all DLS combinations at the two levels are tested at multiple snapshots of the simulation, as listed in Table 11.1. We only show the results of Evrard collapse at time-step 500 as an example. After testing all DLS combinations at different simulation stages, the identified best two-level DLS combination is used to execute the full SPH simulation and measure the achieved overall performance improvement. It is worth noting that the observed load imbalance in Figure 11.3 for a single time-step of SPHYNX is repeatedly be observed through the execution of full SPH simulations, which typically requires $10^5 - 10^6$ time-steps.

#### 11.2.1.3 Two-level dynamic load balancing

Six loop scheduling techniques are considered at the thread level via the *eLaPeSD* OpenMP library and eleven loop scheduling techniques at the process level via the *DLS4LB*, yielding a combination of $6 \times 11 = 66$ experiments per application or test-case. The FSC technique is not considered at the thread level nor the process level as it requires profiling of the application to estimate the standard deviation of task execution times $\sigma$ and the scheduling overhead $h$. Application performance with FSC is significantly influenced by the provided $\sigma$ and $h$ values. Also, the WF technique is not considered at both levels, as it is designed for highly heterogeneous computing systems, which is not the case for the experiments performed herein. Only the Broadwell partition of the miniHPC is considered.

At the process level, the SS technique is not considered, as it assigns a single task to a requesting process, which limits the thread level parallelism as only one thread. Finally, the AWF technique is only used with SPHYNX as it only applies for time-stepping applications, such as SPHYNX.

NODLB and STATIC denote the scenario where application tasks are statically and equally divided among processes or threads, respectively, where each process is assigned $N/P$ particles. Minimum chunk size is specified at the process level to avoid processes being assigned a very small chunk of tasks towards the end of the execution, which will not contain enough work to distribute to threads within a process and increase the scheduling rounds and consequently the overall scheduling overhead. The minimum chunk size at the process level is set to half the chunk size of the mFSC technique, that is 532, 700, 7278, 2549, for Mandelbrot, PSIA, SPHYNX with stellar collision test, and




(a) Mandelbrot, 40 processes, 10 threads

(b) PSIA, 40 processes, 10 threads

(c) Stellar collision time-step 6916, SPHYNX, 40 processes, 10 threads

(d) Evrard time-step 516, SPHYNX, 12 processes, 10 threads

**Figure 11.3 Impact of two-level load imbalance at thread level and process level in the three scientific applications.** Idle time due to load imbalance is shown in yellow at the thread level and in red at the process level.

SPHYNX with Evrard collapse test, respectively. At the thread level, the minimum chunk size is set to 1 loop iteration (OpenMP default) to achieve the best possible load balance as the scheduling overhead is small.

#### 11.2.1.4 Computing system

The two-level dynamic load balancing approach is tested on miniHPC. Recall that each node of miniHPC contains two CPU sockets and ten cores per socket (see Table 8.1). Each MPI rank is pinned to a CPU socket of the miniHPC to improve the data locality among threads within an MPI rank. The number of threads per MPI rank is set to be equal to the number of cores per socket. Therefore, each compute node of miniHPC executes two MPI ranks, one per socket, with 10 OpenMP threads within each MPI rank. 20 nodes of miniHPC are used to test the performance of Mandelbrot, PSIA, and the stellar collision test with SPHYNX, whereas only 6 nodes are used in the Evrard collapse test with SPHYNX. Computing system sizes are chosen such that it results in a reasonable amount of work per core and, therefore, computation to communication ratio.

### 11.2.2 Performance Results and Discussion

The performance of the three scientific applications of interest is reported in Figures 11.7 to 11.9. The figures show the parallel execution time of Mandelbrot and PSIA and the computing time of the gravity computation per time-step for SPHYNX. Figures 11.7 to 11.9 show the percent performance improvement with DLS techniques normalized to not using any dynamic load balancing mechanism at the thread level nor at the process level (NODLB+STATIC). The percent performance improvements are color-coded such that white is the reference baseline, blue shades represent performance improvement while red shades represent performance degradation.

Using two-level dynamic load balancing improved the performance of Mandelbrot by up to 21% compared to the original no dynamic load balancing execution time, as shown in Figure 11.7(a) with TSS and SS jointly at the process level and the thread level, respectively. The performance improvement is much lower in PSIA than in Mandelbrot as PSIA is mildly imbalanced as shown in Figure 11.3(b). Two-level dynamic load balancing improved the performance of the gravity calculations in SPHYNX also by 11% for stellar collision test-case in Figure 11.7(c) with AWF-C at the process level and FAC at the thread level and by 43% for Evrard collapse test-case in Figure 11.8(a) with GSS at the process level and FAC at the thread level. Even though dynamic scheduling incurs higher overhead than static, the balanced load execution improved applications' performance by up to 21%.

The SS technique results in poor performance for SPHYNX at the thread level due to the fine granularity of its tasks (240 microseconds on average). At the process level, the AF technique performs poorly for the experiments conducted herein. Poor AF performance is attributed in part to its significant overhead and the lack of high variability (in application and computing system) to provide proper mitigation between this overhead and the benefit from AF. Specifically, the AF technique is designed for highly irregular workloads that execute in stochastic environments.

Figures 11.4 and 11.5 show the best average parallel execution time over 20 repetitions (Mandelbrot and PSIA) or time-step (SPHYNX) when:

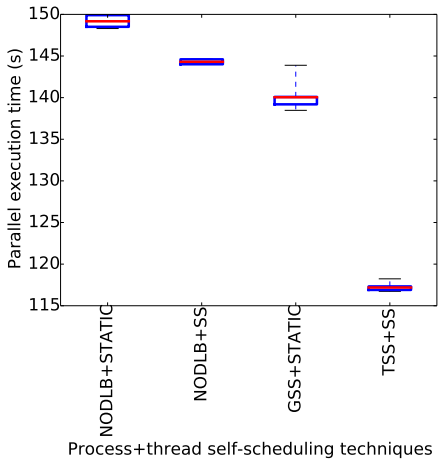
1. Using no load balancing at any of the two levels
2. Using the best DLS technique only at the thread level
3. Using the best DLS technique only at the process level
4. Using the best available combination of DLS techniques at the thread level and the process level

The results in Figures 11.4 and 11.5 show the benefits of two-level dynamic load balancing versus single-level alone, as conceptually illustrated in Figure 11.2. The results confirm that certain performance gains are achieved by single-level dynamic load balancing (either thread level or process level, middle boxes) as predicted by Figure 11.2, however, the best performance is always achieved by two-level dynamic load balancing.

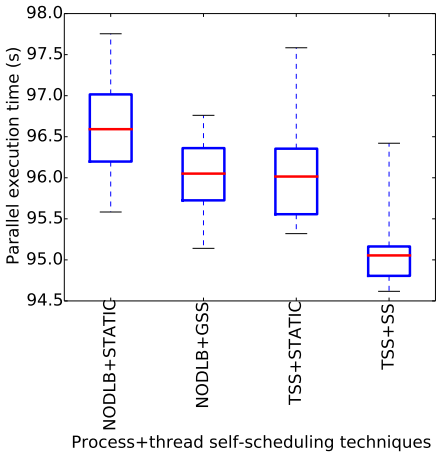
GSS+FAC was identified as the best combination of DLS techniques at the process level and the thread level, respectively, by testing all 66 DLS combinations at the two levels for the Evrard collapse test at different stages of the simulation, namely at time-steps 100, 500, 1000, 1700, 2000, 2300, 2500, and 2800. Figure 11.6 shows the parallel execution time per time-step for the full simulation of Evrard collapse with one million particles. The results show that the execution time per time-step with two-level dynamic load balancing is always better than the baseline with no load balancing at either of the two levels and achieves an overall application performance improvement of 15%.

#### 11.2.2.1 Discussion

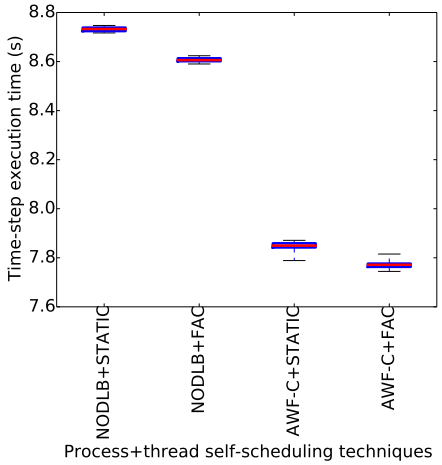
The execution traces in Figure 11.3 shows different profiles of the two-level load imbalance in the three applications. The Mandelbrot execution trace in Figure 11.3(a) shows a severe case of two-level load imbalance, where there is high variability in process and thread (within a single process) finishing times. Single-level dynamic load balancing



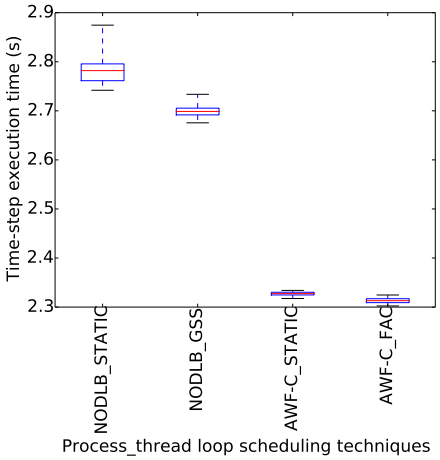

(a) Mandelbrot, 40 processes, 10 threads



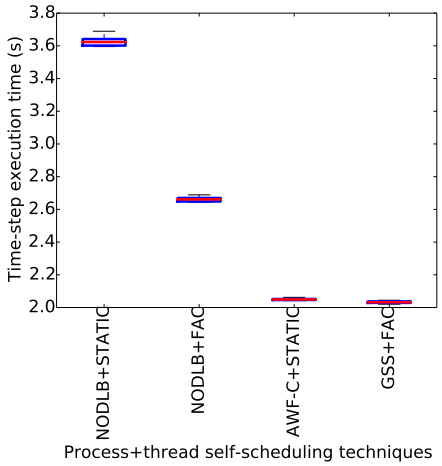

(b) PSIA, 40 processes, 10 threads



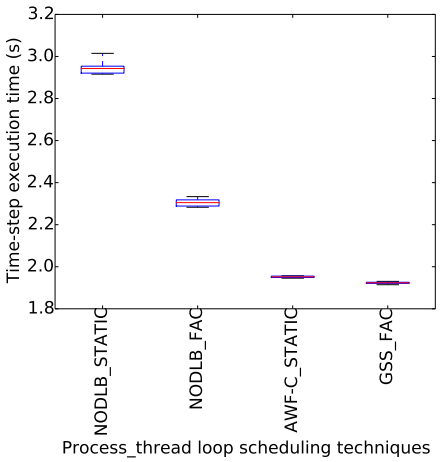

(c) Stellar collision time-steps 6900-6920, 40 processes, 10 threads




(d) Evrard collapse time-steps 100-120, 12 processes, 10 threads

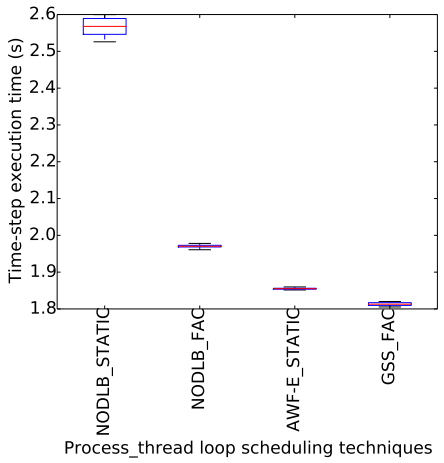



(e) Evrard collapse time-steps 500-520, 12 processes, 10 threads

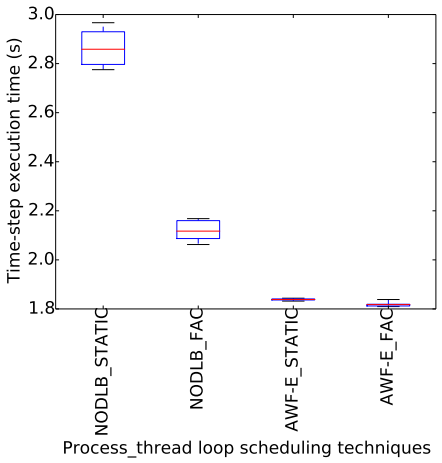



(f) Evrard collapse time-steps 1000-1020, 12 processes, 10 threads

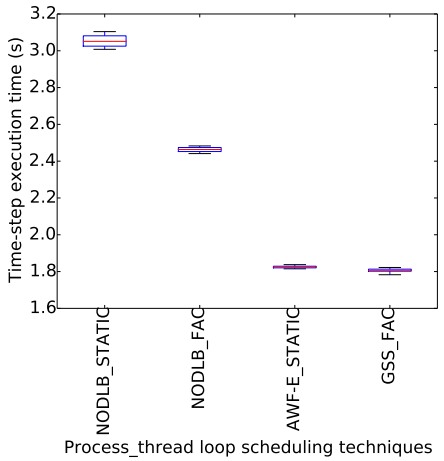
**Figure 11.4 Impact of single- and two-level dynamic load balancing on the execution time of the three scientific applications.** Each plot shows in the following order: the execution time with the baseline (NODLB+STATIC), best DLS technique at thread level, best DLS technique at process level, and best two-level DLS combination. The red line represents the average performance computed over 20 repetitions or time-steps, the boxes define the first and third quartiles, and the whiskers are maximum and minimum values.



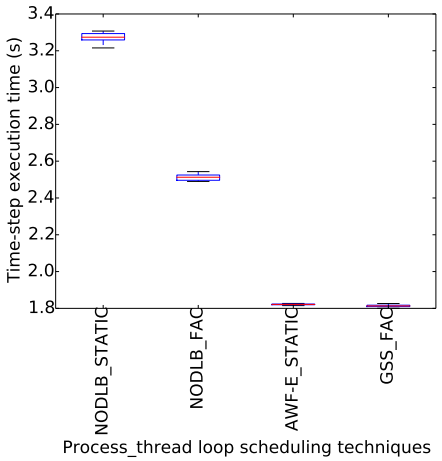

(a) Evrard collapse time-steps 1700-1720, 12 processes, 10 threads



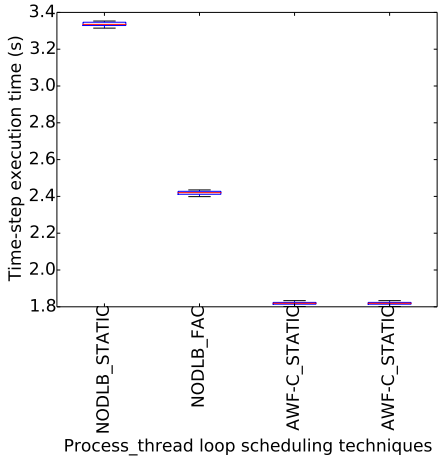

(b) Evrard collapse time-steps 2000-2020, 12 processes, 10 threads




(c) Evrard collapse time-steps 2300-2320, 12 processes, 10 threads




(d) Evrard collapse time-steps 2500-2520, 12 processes, 10 threads




(e) Evrard collapse time-steps 2800-2820, 12 processes, 10 threads

**Figure 11.5 Impact of single- and two-level dynamic load balancing on the execution time of the three scientific applications.** Each plot shows in the following order: the execution time with the baseline (NODLB+STATIC), best DLS technique at thread level, best DLS technique at process level, and best two-level DLS combination. The red line represents the average performance computed over 20 repetitions or time-steps, the boxes define the first and third quartiles, and the whiskers are maximum and minimum values.

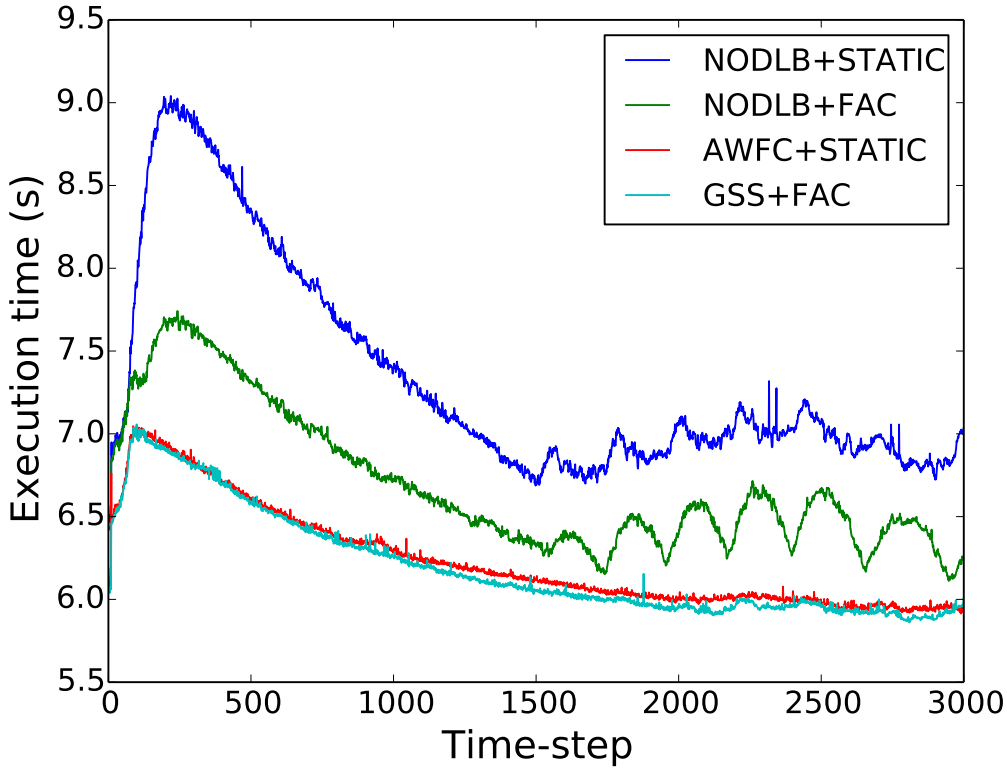



**Figure 11.6** **Time-step execution time of the Evrard collapse test-case in SPHYNX with 12 processes and 10 threads per process.** The two-level dynamic load balancing improved the performance of SPHYNX throughout the full simulation (0 : 3000 time-step), achieving 15% overall improvement in execution time.

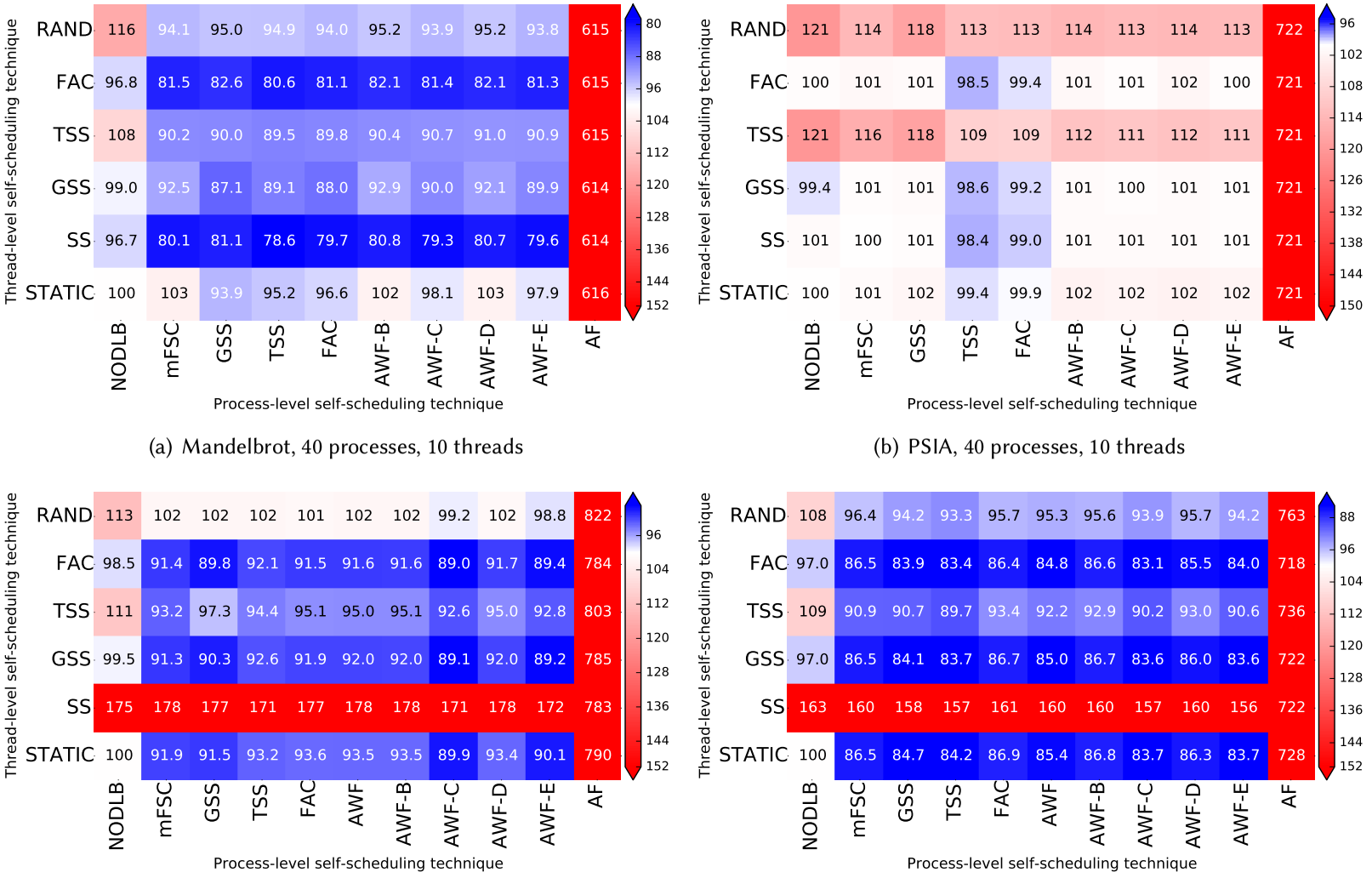
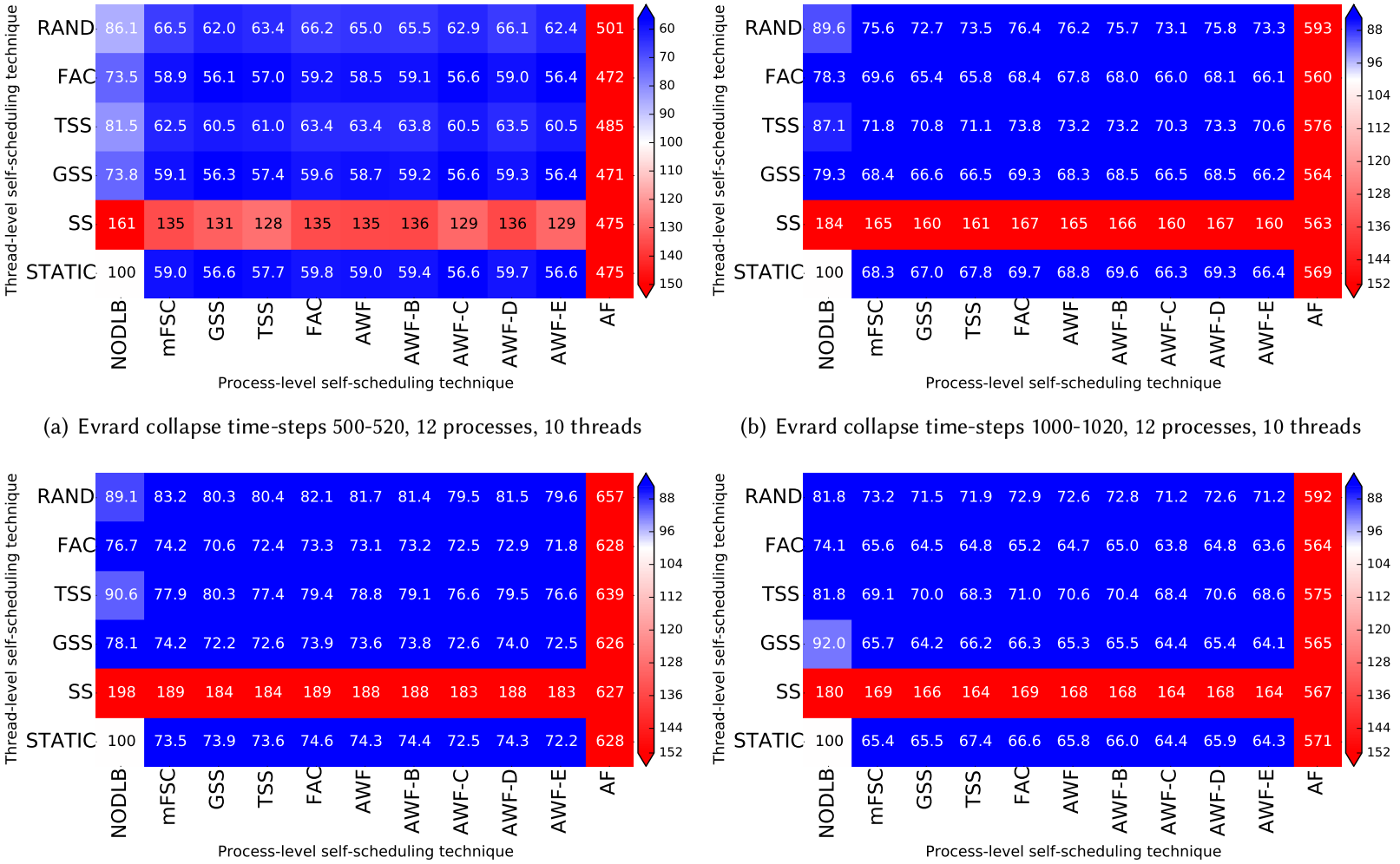



**Figure 11.7** **Impact of the two-level dynamic load balancing on the performance of the three scientific applications.** Percent improvement corresponds to the average of 20 repetitions or time-steps with two-level load balancing normalized compared to NODLB+STATIC. White, red, and blue correspond to baseline, degraded, and improved performance, respectively.



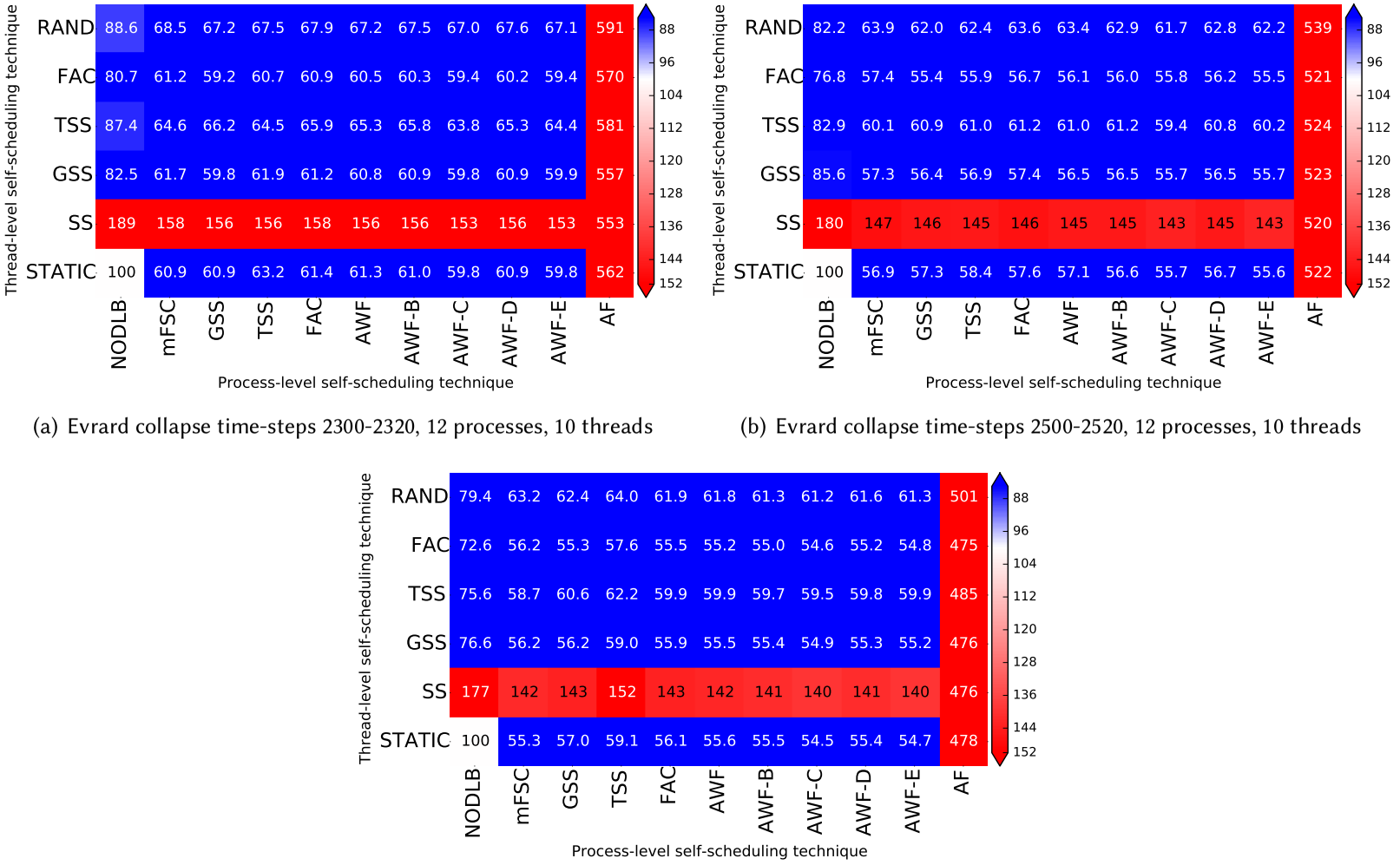

(a) Evrard collapse time-steps 500-520, 12 processes, 10 threads

(b) Evrard collapse time-steps 1000-1020, 12 processes, 10 threads

(c) Evrard collapse time-steps 1700-1720, 12 processes, 10 threads

(d) Evrard collapse time-steps 2000-2020, 12 processes, 10 threads

**Figure 11.8 Impact of the two-level dynamic load balancing on the performance of the three scientific applications.** Percent improvement corresponds to the average of 20 repetitions or time-steps with two-level load balancing normalized to NODLB+STATIC. White, red, and blue correspond to baseline, degraded, and improved performance, respectively.




(a) Evrard collapse time-steps 2300-2320, 12 processes, 10 threads

(b) Evrard collapse time-steps 2500-2520, 12 processes, 10 threads

(c) Evrard collapse time-steps 2800-2820, 12 processes, 10 threads

**Figure 11.9 Impact of the two-level dynamic load balancing on the performance of the three scientific applications.** Percent improvement corresponds to the average of 20 repetitions or time-steps with two-level load balancing normalized to NODLB+STATIC. White, red, and blue correspond to baseline, degraded, and improved performance, respectively.

(thread or process level) in this case only achieves limited performance improvement, as shown in Figure 11.2. This is experimentally confirmed in Figure 11.4(a), which indicates the significant improvement of performance with two-level dynamic load balancing versus its slight improvement with a single-level load balancing.

While Mandelbrot represents an extreme case of two-level load imbalance, PSIA represents the other extreme, where processes and threads within a process are only slightly imbalanced, as shown in Figure 11.3(b). Therefore, the maximum achievable performance improvement in PSIA is much smaller (1.6%) than the one in Mandelbrot (21.4%).

The SPHYNX execution with stellar collision suffers from high load imbalance at the process level and low load imbalance at the thread level (Figure 11.3(c)). This is reflected by a slight effect of thread level load balancing and a significant impact of the process level load balancing in improving its performance (Figure 11.4(c)).

For the Evrard collapse, one can observe severe load imbalance at the process level. This is caused by two threads lagging the execution of the last process significantly behind all other processes (Figure 11.3(d)). In this case, thread-level dynamic load balancing not only improves the load balancing at the thread level but also at the process level, as the last process will finish earlier, thus closer to the other processes finishing times. Alternatively, process-level dynamic load balancing will also distribute the high workload of the last two threads among all the processes, therefore, dissolving the load imbalance due to the lagging threads among processes. This is confirmed by the significant performance improvement achieved by dynamic load balancing at thread-level alone and at process-level alone in Figure 11.4(e). This represents an interesting case where dynamic load balancing from the thread level propagates to the process level and vice versa.

We note that applying thread-level load balancing requires a minimum implementation effort (especially with the OpenMP scheduling clause as shown in Algorithm 11.1). Therefore, going from single-level process-level load balancing to two-level dynamic load balancing is almost effortless and always achieves the best performance based on the experiments conducted herein.

Also, *dynamic load balancing at thread level and process level influence each other's performance, and this influence should not be ignored.* For example, the best combination of DLS techniques at the thread and process level (fourth column), is not always a combination of the best technique at the thread level (second column) with the best technique at the process level (third column) in Figures 11.4 and 11.5. In certain cases, *dynamic load balancing at thread level propagates to the process level and vice versa.* Consequently, *the best-performing two-level combination is not always the combination of the two best performing DLS techniques at a single level alone.* These observations reveal the *significant interplay* between thread level and process level dynamic load balancing, as

load balancing at one level influences the load imbalance at the other level.

Part IV

# Robust Scheduling

# 12

# SimAS: Simulation-assisted DLS Algorithm Selection for Irregular Systems Performance

Scientific applications' performance on HPC systems degrades due to load imbalance and perturbations. In Chapter 3, we showed that applications on large-scale HPC systems are often perturbed due to nonfatal errors and interference of system services (Section 3.5). DLS techniques improve applications' performance by balancing their load during the execution.

Given the broad range of DLS techniques and their properties (Section 2.2), the choice of the DLS technique that improves an application performance in uncertain execution scenarios due to perturbations is challenging. This defines the algorithm selection problem [Ric76] in the context of scheduling. In this chapter, we answer the research question *"How to select a DLS technique that will achieve improved performance under perturbations?"*. We devised SimAS, where realistic performance simulations are leveraged for the dynamic selection of the most efficient DLS techniques during execution according to application and system state and perturbations (see Figure 12.1).

The study of the performance of scientific applications with DLS under perturbations revealed that the most robust DLS technique, identified as the DLS technique that results in the least variation of the application execution time under various perturbations, does not achieve the best performance in all execution scenarios [MC18a]. Figure 12.2 shows the simulative performance of PSIA (c.f. Section 12.2.1.1) on 696 cores of miniHPC (c.f. Section 12.2.1.4) under perturbations (c.f. Section 12.2.1.6).

According to these results, GSS is the most robust DLS technique due to the minimal variability of its performance under perturbations (Figure 12.2(a)). However, it results in poor application performance under perturbations. Even the second most robust DLS

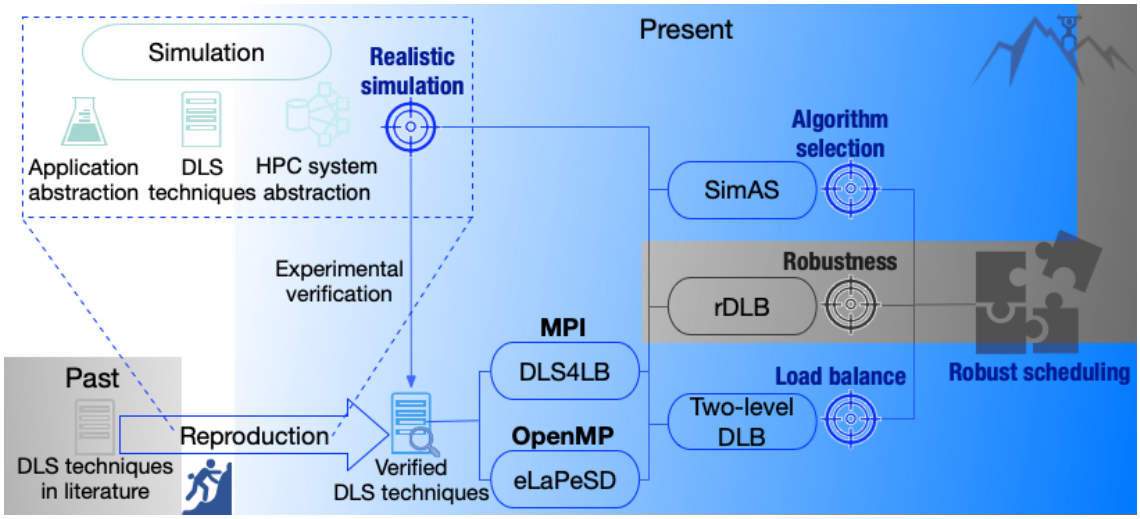
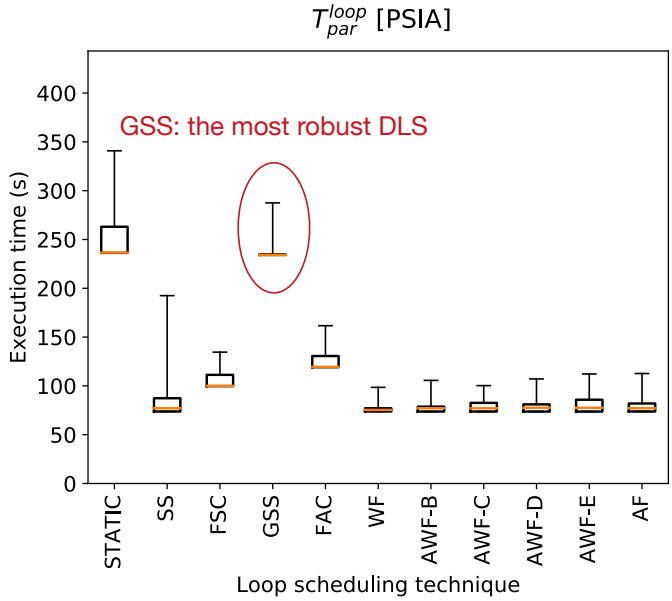



**Figure 12.1 Illustration of the focus and progress in this chapter (in colors) as part of the overall approach (in grayscale).** Realistic simulations are leveraged to address the DLS algorithm selection problem under perturbations.

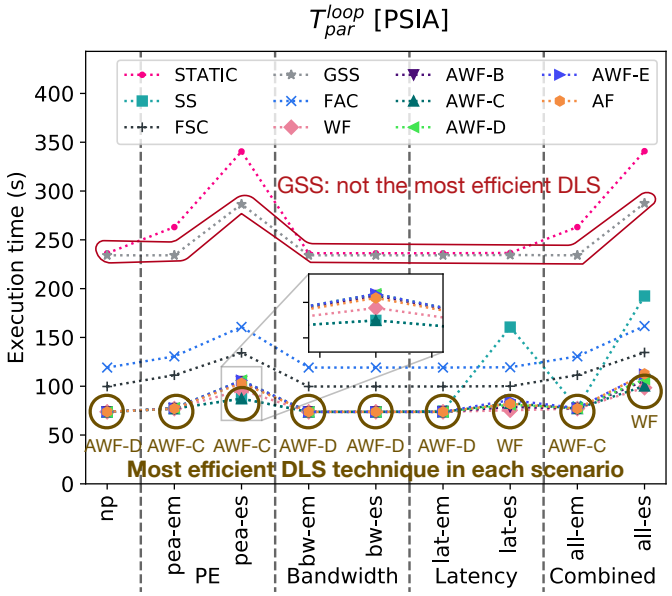
technique, WF, is outperformed by SS and AWF-C in specific perturbation scenarios, as can be seen in Figure 12.2(b). These results suggest that even if the most robust DLS technique is known a priori, which is hard to achieve, the application performance is degraded in different execution scenarios. Therefore, a methodology for the dynamic selection of DLS techniques during execution is needed to achieve the highest possible performance in all execution scenarios. In this chapter, we present an approach, Simulation-assisted scheduling Algorithm Selection (SimAS), to address the problem of scheduling algorithm selection problem for scientific applications executing on HPC systems under perturbations.

## 12.1 Simulation-assisted DLS Selection

The SimAS concept is motivated by the well-known control strategy model predictive control (MPC) [Raw00] where different control signals are tested on a model of the controlled system before it is applied to the real control system. The MPC controller *predicts* the performance of the system with different control signals to optimize system performance. Employing the same concept in task scheduling, SimAS uses simulation to *predict* the application performance with different DLS techniques (control signals) and *selects* the DLS technique that achieves the highest performance for the given application, system, and perturbations in the system. As shown in Figure 12.3, a call to a loop simulator is inserted inside a typical scheduling loop. SimAS leverages loop simulators to predict the performance of the application with various DLS techniques. The




(a) Variation of perfomance under perturbations.




(b) Performance under perturbations.

**Figure 12.2** **Impact of perturbations on applications performance with DLS.** The simulative performance of PSIA (c.f. Section 12.2.1.1) under perturbations in computing availability, network bandwidth, and latency (c.f. Section 12.2.1.6). The most robust DLS technique which incurs the least performance variation under various perturbations Figure 12.2(a) is GSS. As shown in Figure 12.2(b), GSS does not achieve the best performance under all perturbations [MC18a].

system monitor and estimator components read the system state during the execution and update the computing system representation accordingly to feed the simulator with the current perturbations in the system. SimAS examines the predicted performance by

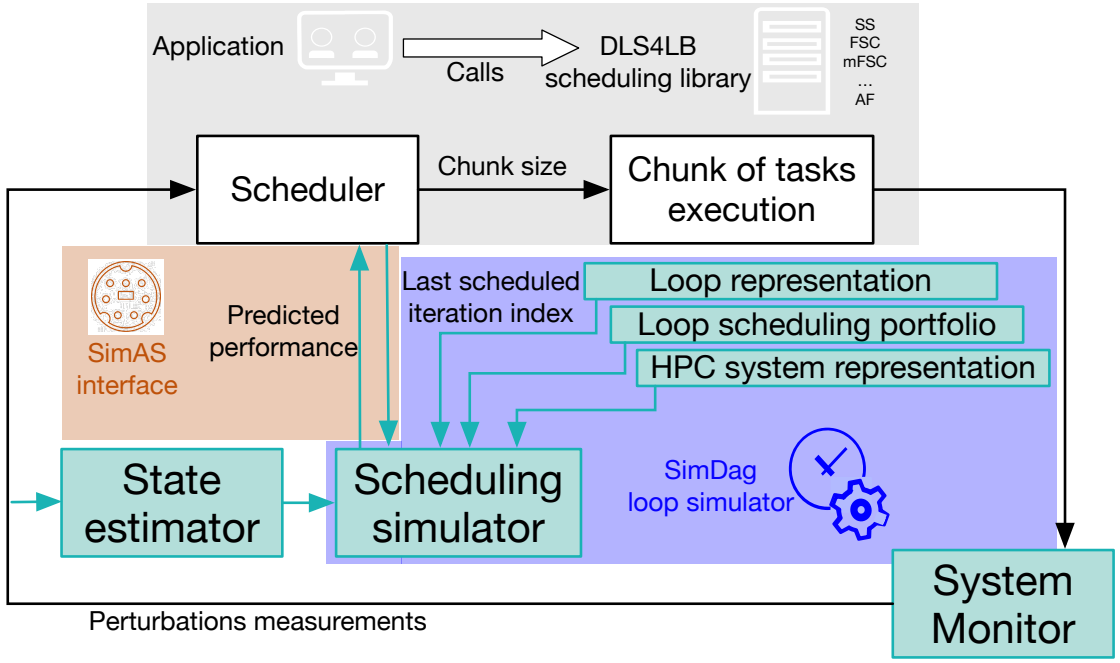



**Figure 12.3 The proposed *simulation-assisted scheduling algorithm selection* (SimAS) approach.** The components highlighted in mint color represent the SimAS additions to a typical loop scheduling system.

the simulator with different DLS techniques and *selects* the DLS technique that achieves the best performance in the current execution scenario. The above steps may be repeated several times during the execution of the loop, and the SimAS call frequency can be aligned with the frequency or intensity of the perturbations. The main idea of the SimAS (inspired by MPC controller) is to use the simulator (system model) to test different DLS techniques (control signals) on the loop execution (the system), before actually applying the selected DLS to the executing application.

The advantage of SimAS is that it leverages the use of already developed state-of-the-art simulations [MEC+18a; MEC+19] to predict the performance dynamically during execution. The prediction accuracy of a simulator is strongly influenced by the representation of both the applications and the systems in simulation as well as by the subsystem models it comprises. Since loop simulators predict the performance of load imbalanced computationally-intensive loops, the influence of the memory subsystem (e.g., complex memory hierarchy) on their performance is minimal. Therefore, application performance can accurately be predicted via simulation. For instance, the percent error between native and simulative executions for a given application (PSIA [EMC17a]) using the SG-SD interface was found to be between 0.95% and 2.99% [MEC+18a]. It is expected that the accuracy and speed of the simulators employed by SimAS will improve as simulators, in general, are continuously being developed and refined. The cost of frequent calls to SimAS can be amortized by launching parallel sim-

ulator instances to concurrently derive predictions for various DLS. Alternatively, this cost can be entirely mitigated by asynchronously calling SimAS, concurrently to the application execution. The system monitor and estimator components can be implemented with several system monitoring tools [Cio17], such as `collectl`. Such tools can periodically be instantiated to measure PE and network loads and to update the system representation in the simulator for the next call to SimAS. The measured chunk execution times can also be used to estimate the current PE computational speeds. The PE loads can be estimated and predicted using autoregressive integrated moving average [MBS+15]. The implementation details of the loop simulator and SimAS are described below in Section 12.2.

## 12.2 Experimental Evaluation

The SimAS approach is evaluated using an extensive set of experiments employing native and simulative experimentation, real and synthetic, single-sweep, and time-stepping applications, on heterogeneous cores of miniHPC system (c.f. Section 8.2.1).

### 12.2.1 Design of Experiments

The design of the factorial experiments is presented in the following (cf. Table 12.1), along with details of the DLS techniques and SimAS implementation, the implemented loop simulator, the computing system under test and its injected perturbations in native and simulative experiments.

#### 12.2.1.1 Applications

We consider two applications (executed as single-sweep applications and as time-stepping applications) and five synthetic (single-sweep) workloads.

**Real applications** *1. PSIA.* The parallel spin-image algorithm [EMC17a] (PSIA), is a computationally-intensive application from computer vision (see Section 8.1). The total number of parallel loop iterations in PSIA is $400,000$.
*2. Mandelbrot.* This application computes the Mandelbrot set [Man80] and generates its image. Mandelbrot is parallelized such that the calculation of the value at every single pixel of a 2D image is a loop iteration that is performed in parallel (see Section 8.1).
*3. PSIA_TS.* This application similar to PSIA. Unlike PSIA, PSIA_TS is executed in time-steps, where at each time-step a certain number ($4,000$) of spin-images is generated from a 3D object. It simulates applying spin-image transformations to an object

**Table 12.1 Design of factorial experiments**

| Factors | Values | Properties |
|---|---|---|
| **Applications** | PSIA | $[5.9 \cdot 10^7$ .. $6.6 \cdot 10^7]$ FLOP per iteration |
| | Mandelbrot | $[5.9 \cdot 10^1$ .. $2.6 \cdot 10^8]$ FLOP per iteration |
| | PSIA_TS (time-stepping) | $[5.9 \cdot 10^7$ .. $6.5 \cdot 10^7]$ FLOP per iteration |
| | Mandelbrot_TS (time-stepping) | $[5.9 \cdot 10^1$ .. $2.6 \cdot 10^8]$ FLOP per iteration |
| | Constant | $2.3 \cdot 10^8$ FLOP per iteration |
| | Uniform | $[10^3$ .. $7 \cdot 10^8]$ FLOP per iteration |
| | Normal | $\mu = 9.5 \cdot 10^8$ FLOP, $\sigma = 7 \cdot 10^7$ FLOP, $[6 \cdot 10^8$ .. $1.3 \cdot 10^9]$ FLOP per iteration |
| | Exponential | $\lambda = 1/3 \cdot 10^8$ FLOP, $[9.48 \cdot 10^2$ .. $4.5 \cdot 10^9]$ FLOP per iteration |
| | Gamma | $k = 2$, $\theta = 10^8$ FLOP, $[4.1 \cdot 10^6$ .. $2.7 \cdot 10^9]$ FLOP per iteration |
| | Problem size | $N = 400{,}000$ iterations (all applications except for<br>PSIA_TS, $N = 4{,}000$ iterations per time-step $\times 10$ time-steps<br>Mandelbrot, $N = 262{,}144$ iterations<br>Mandelbrot_TS, $N = 16{,}384$ iterations per time-step $\times 10$ time-steps) |
| **Loop scheduling** | STATIC | Static |
| | SS, FSC, mFSC, GSS, TSS, FAC, WF | Dynamic nonadaptive |
| | AWF-B, -C, -D, -E, AF | Dynamic adaptive |
| **Computing system** | miniHPC<br>(heterogeneous HPC cluster) | 22 Intel Broadwell nodes ($22 \cdot 20$ cores), relative core weight[1] $= 0.817$<br>4 Intel Xeon Phi KNL nodes ($4 \cdot 64$ cores), relative core weight[1] $= 0.183$<br>$P = 128$ heterogeneous ($4 \times 16$ Broadwell + $1 \times 64$ KNL) cores |
| **Perturbations** | Nominal conditions | `np` (no perturbations) |
| | PE availability | `pea-cm` (constant mild): $\mu = 75\%$, $\sigma = 0\%$<br>`pea-cs` (constant severe): $\mu = 25\%$, $\sigma = 0\%$<br>`pea-em` (exponential mild): $\mu = 78\%$, $\sigma = 24 \cdot 10^{-3}\%$<br>`pea-es` (exponential severe): $\mu = 31\%$, $\sigma = 89 \cdot 10^{-3}\%$ |
| | Bandwidth | `bw-cm` (constant mild): $\mu = 1 \cdot 10^{-5}\%$, $\sigma = 0\%$<br>`bw-cs` (constant severe): $\mu = 1 \cdot 10^{-7}\%$, $\sigma = 0\%$<br>`bw-em` (exponential mild): $\mu = 1.1 \cdot 10^{-1}\%$, $\sigma = 9 \cdot 10^{-2}\%$<br>`bw-es` (exponential severe): $\mu = 23 \cdot 10^{-2}\%$, $\sigma = 19 \cdot 10^{-2}\%$ |
| | Latency | `lat-cm` (constant mild): $\mu = 1 \cdot 10^{-5}\%$, $\sigma = 0\%$<br>`lat-cs` (constant severe): $\mu = 1 \cdot 10^{-7}\%$, $\sigma = 0\%$<br>`lat-em` (exponential mild): $\mu = 1.2 \cdot 10^{-5}\%$, $\sigma = 1.5 \cdot 10^{-5}\%$<br>`lat-es` (exponential severe): $\mu = 2.9 \cdot 10^{-7}\%$, $\sigma = 1.8 \cdot 10^{-7}\%$ |
| | Combined | `all-cm` (constant mild): `pea-cm`, `bw-cm`, and `lat-cm`<br>`all-cs` (constant severe): `pea-cs`, `bw-cs`, and `lat-cs`<br>`all-em` (exponential mild): `pea-em`, `bw-em`, and `lat-em`<br>`all-es` (exponential severe): `pea-es`, `bw-es`, and `lat-es` |
| **Experimentation** | Native[2] | PSIA, Mandelbrot, PSIA_TS, and Mandelbrot_TS<br>on 128 miniHPC cores under targeted perturbations |
| | Simulative | PSIA and Mandelbrot on 128 miniHPC cores under all perturbations<br>Synthetic applications on 128 miniHPC cores under all perturbations |

[1] Relative core weight to the total speed of a system of one core of each type.
[2] Direct experiments on real HPC systems.

in motion (a video), where at each time-step a certain number of spin-images is created. PSIA_TS is executed for 10 time-steps.

*4. Mandelbrot_TS.* This is the time-stepping version of Mandelbrot application. At each time-step, the generated Mandelbrot set image at time $t$ is zoomed-in by 5% on the center of the image to generate the image at $t+1$. Mandelbrot_TS is executed for 10 time-steps. The workload per time-step is reduced compared to Mandelbrot (single-sweep) such that the execution time of 10 time-steps of Mandelbrot_TS is comparable to the execution time of the single-sweep version. This is desirable for native experimentation given the broad set of experiments performed (see Table 12.1), to avoid extremely long execution times.

**Synthetic workloads** Five synthetic workloads are examined herein. Each of the five synthetic workloads contains $400,000$ parallel loop iterations. The number of floating-point operations (FLOP count) in each loop iteration is assumed to follow five different probability distributions, namely: constant, uniform, normal, exponential, and gamma probability distributions. This assumption captures the characteristics of a wide range of applications [BBC+17]. Synthetic applications are used to cover a broader spectrum of load imbalance profiles than what may be encountered in real applications. The probability distribution parameters used to generate these FLOP counts are also given in Table 12.1.

#### 12.2.1.2 Loop scheduling

Thirteen loop scheduling techniques are used to assess the performance of the above nine applications under various execution scenarios. These techniques represent a wide range of *static* and *dynamic* loop scheduling approaches. The dynamic loop scheduling (DLS) techniques can further be distinguished into five adaptive and seven nonadaptive techniques. Algorithm 12.1 shows, in blue font color, the lines needed to be added to the application code to parallelize it with *DLS4LB*.

#### 12.2.1.3 Simulation-assisted scheduling algorithm selection approach (SimAS)

The *DLS4LB* is extended to support the SimAS as the fourteenth scheduling option in the *DLS4LB*. Taking the same approach of the *DLS4LB* of minimal application code changes, an application can use the SimAS by inserting only two function calls, shown in green font color in Algorithm 12.1.

The `SimAS_setup` function sets up the primary data structure `SimAS_info` that holds important information, such as the number of PEs, the number of loop iterations, the path to the simulator, the FLOP file that contains the FLOP count per loop iteration, and the platform file that describes the computing system. Also, `SimAS_setup` asynchronously starts the simulation of the application performance immediately with a portfolio of DLS techniques in parallel. The `SimAS_setup` sets the scheduling technique to a default DLS (WF herein, which is selected based on the simulation results) to allow the application to start and avoid delaying the application execution.

The `SimAS_update` checks (every 5 seconds herein) if the simulation is finished, and selects the DLS technique that allows the application to finish the largest number of tasks in the shortest time by manipulating the DLS_info structure; otherwise, it will keep the selected DLS technique unchanged. The `SimAS_update` reruns the simulation again if 50 seconds (the SimAS calling frequency and it can be customized by

`SimAS_setup`) have passed since the simulator was previously called or every new time-step for time-stepping applications. The `SimAS_update` prevents the start of a new instance of the simulator unless the earlier one is completed or the number of remaining unscheduled iterations is less than or equal the number of PEs.

**SimAS improvements** Several measures are taken to mitigate the overhead of simulation during execution, such as simulating in parallel, asynchronously to the application, to avoid stopping the application execution. A default DLS technique (namely, WF) was used until the simulation completes, based on its high performance in the simulative experiments. The SimAS checks the completion of the simulation at specific (adjustable) periods to reduce the overhead of these checks. DLS techniques with poor performance on heterogeneous computing systems were excluded from the DLS portfolio provided to the SimAS to reduce the number of needed simulations and the search space of the SimAS. Simulations launched by the SimAS were executed on the four cores per node that are not used by the application and left for OS, network, and other processes (see Section 12.2.1.4). Therefore, running simulations do not perturb the executing application.

**Algorithm 12.1 Dynamic load balancing with SimAS support using *DLS4LB***

```
#include <mpi.h>
#include "DLS4LB.h"
#include "SimAS.h"
MPI_Init(&argc, &argv)
MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
MPI_Comm_rank(MPI_COMM_WORLD, &myid)
scheduling_method = SimAS_setup(SimAS_info, P, N, h, sigma, sim_path, FLOP_file,
platform_file)
DLS_Setup(MPI_COMM_WORLD, DLS_info)
DLS_Start_loop (DLS_info, N, scheduling_method)
while (! DLS_Terminated(DLS_info)) do
    SimAS_update(DLS_info, SimAS_info)
    DLS_Start_chunk(DLS_info, Cstart, Csize)
    Compute_iterations(Cstart, size)
    DLS_End_chunk(DLS_info)
DLS_End_loop(DLS_info)
MPI_Finalize()
```

#### 12.2.1.4 Computing system

The native experiments are conducted on miniHPC (see Section 8.2.1). Four Broadwell nodes and four KNL nodes of the miniHPC are used to create a heterogeneous system

of 128 cores (see Table 12.1). A heterogeneous system is selected for the experiments as it is more challenging, load balancing wise, and perturbing the application execution on this system presents another load balancing challenge. All nodes are interconnected with Intel Omni-Path fabrics in a nonblocking two-level fat-tree topology. To ensure the uninterrupted execution of the application on the allocated CPU cores, only 16 out of the 20 cores per node (8 out of 10 per socket) are used. The other 4 cores per node are left for operating system, Omni-Path network load, and other non-application-related processes.

#### 12.2.1.5 Simulation details

**Applications** The SG-SD (see Section 4.1) is used to simulate the applications of interest, where the loop iterations in the application code are represented as tasks [MEC+18a]. The number of floating-point operations (FLOP) of each loop iteration is counted using PAPI counters [BDG+00] to represent the computational effort associated with an application's loop iterations. The SG-SD then reads the FLOP count per iteration during execution to simulate the computation per iteration. All DLS techniques supported by the *DLS4LB* are also implemented in the SG-SD, and tasks are assigned to free and requesting simulated cores, similar to native execution. The pseudocode of the loop simulator is presented in Algorithm 6.1. The implemented loop simulator reads in the number of iterations (tasks), start task ID, the path to the file that contains the FLOP count per loop iteration, the path to the computing system representation, the selected scheduling technique, and the maximum *simulated time*. The simulator reads the data and simulates the loop execution using the selected DLS technique. The simulator then outputs the simulated time and the number of tasks executed in this simulated time. This information is read by the SimAS, which compares different DLS techniques based on this information and selects the DLS technique that results in the shortest execution time and the largest number of finished tasks.

**Computing system** A computing system is represented in SG via an XML file denoted as `platform file` (see Section 6.3). The computational speed of a PE is estimated by measuring a loop execution time and dividing it by the total number of floating-point operations included in the loop [MEC+18a]. A Broadwell core was found to be four times faster than a KNL core, as indicated by the relative core weights (cf. Table 12.1).

#### 12.2.1.6 Injected perturbations

Based on the observations of the nonfatal perturbations on real systems and the considered perturbation in previous robust scheduling studies (see Chapter 3), three cat-

egories of perturbations are considered herein, namely *delivered computational speed*, *available network bandwidth*, and *available network latency*. Two intensities are considered, `mild` and `severe`, for each category. Two scenarios are considered for each intensity, where the value of the delivered computational speed is either `constant` or `exponentially` distributed.

All perturbations (cf. Table 12.1) are considered to occur periodically, with a period of 100 seconds where the perturbations affect the system only during 50% of the perturbation period. The network (bandwidth and latency) perturbations commence with the application execution, while the delivered computational speed perturbations begin 50 seconds after the start of the application. Another perturbation scenario is created by combining all perturbations from the other individual categories.

**Perturbations in simulative experiments** All perturbations are enacted in SG during simulation via the `availability`, `bandwidth`, `latency`, and `platform` files (see Section 6.3) to represent perturbations in delivered computational speed, network available bandwidth, and network latency, respectively.

**Perturbations in native experiments** A program (CPU burner) is launched in parallel and pinned on the same processor cores as the application to induce perturbations on the PE availability in native execution. The program is executed periodically every 100 seconds and is only active during a fraction of this period that corresponds to the required PE availability perturbation (75% or 25%).

For injecting perturbations in the link latency, the MPI communication functions are intercepted using the MPI profiling interface (PMPI), and certain delays are inserted to simulate longer communication latencies. Given that the applications of interest are computationally-intensive and the communicated data size between application's processes is minimal, perturbations in the network bandwidth do not have a significant effect on the application performance, which is confirmed by the simulative experiments below. Therefore, perturbations in the network bandwidth are excluded from native experimentation.

A combined perturbations scenario is created for the native execution by combining PE availability perturbations and network latency perturbations. As both perturbation distributions (constant and exponential) have a comparable effect on the performance, where the impact of constantly distributed perturbations is more evident, only the constant distribution of perturbations is considered in the native experiments.

### 12.2.2 Performance Results

The performance results of the execution of the applications with different loop scheduling techniques under different execution scenarios are illustrated and discussed. We show the need and the importance of the proposed SimAS algorithm selection approach.

#### 12.2.2.1 Simulative performance under perturbations

The simulative performance results of the two applications, PSIA and Mandelbrot, and the five synthetic workloads under perturbations are shown in Figures 12.4 to 12.10. Comparing the performance of the applications with various DLS techniques under no perturbations, i.e., `np`, one can note that STATIC, GSS, TSS, and FAC perform poorly on heterogeneous systems. Their poor performance is because these techniques do not account for the different computational power of different PEs. However, SS, FSC, mFSC, WF, and adaptive techniques improve the performance significantly under no perturbations.

Under perturbations, WF does not accommodate the variability in the PE computing speed due to perturbations as PE weights are constant. The performance of FSC and mFSC is, in general, better than that of STATIC, GSS, TSS, and FAC. However, FSC and mFSC are significantly affected by the perturbations in PEs availability, as well. SS is resilient to perturbations in the delivered computational speed of the PEs. However, it is significantly influenced by the network latency variations, as can be seen with PSIA for example in Figure 12.4 `lat-cs` and `lat-es`.

Perturbations in the network bandwidth show a minimal influence on performance, as the PEs only communicate loop iteration indices to calculate the start index and the size of the next chunk. Therefore, the communicated messages are small. *The bandwidth perturbations are, thus, not selected for subsequent more targeted native experiments under perturbations.* However, network latency perturbations have a significant effect on the performance of applications.

The adaptive techniques perform comparably, with a slight advantage for AWF-E as can be seen in Figure 12.5 `all-cm` and `all-es`. However, in certain cases, such as `lat-cm`, `lat-em`, and `all-em` for in Mandelbrot (Figure 12.5), AWF-B and AWF-D perform significantly poorer than all other techniques. This poor performance is due to the high variation of loop iteration execution times of the Mandelbrot that results in one rank obtaining a chunk that delays the application execution and results in a mis-estimation of PE weights. In general, WF results in the best execution times in most of the considered execution scenarios. However, in some instances, other techniques outperform the WF technique. Specifically, SS outperforms WF in Figure 12.5 `pea-cs`

and `pea-es`.

The results of synthetic workloads also confirm observations from the simulative performance of real applications. *These results suggest that no single DLS outperforms all other techniques in all execution scenarios and even the most robust DLS technique might result in suboptimal performance in certain execution scenarios.* Therefore, the best strategy is to dynamically select a DLS based on the current application and system states.

The SimAS is called every 50 seconds, when there is a work request, to select the best performing DLS. In certain cases, the application performance with SimAS was slightly poorer than the best execution time achieved by other DLS techniques. Poor performance in these cases is because loop scheduling is, by definition, non-preemptive, and the execution of already scheduled loop iterations can not be preempted to be resumed with the newly (expected more suitable) selected DLS.

#### 12.2.2.2 Native performance under perturbations

A targeted selection of native experiments has been conducted for PSIA and Mandelbrot. The constant distribution of perturbation values was selected, as it significantly impacts the performance of the applications. Based on the results of the simulative experiments, perturbations in PE availability and network latency were considered due to their significant influence on applications' performance.

The performance results of PSIA and Mandelbrot with the thirteen DLS techniques under targeted perturbations are shown in Figures 12.11 and 12.12. Similar to the simulation-based predictions in Figures 12.4 and 12.12, the nonadaptive DLS techniques perform poorly on perturbed heterogeneous systems. In particular, STATIC, GSS, TSS, and FAC are significantly affected by all considered perturbations. Unlike in the simulation-based predictions, STATIC is also slightly affected by latency perturbations. The poor performance of STATIC is because it is implemented in the *DLS4LB* in a self-scheduling manner, i.e., workers obtain chunks of loop iterations during execution when they become free. The chunk size of STATIC is equal to the total number of loop iterations divided by the number of worker processes. Therefore, each worker obtains exactly one chunk.

The adaptive techniques resulted in comparable performance. However, in some instances, AF performed poorly in latency perturbation scenarios. Similar to the simulation-based predictions, the WF outperforms all other techniques in most of the execution scenarios. The SimAS results in improved application performance in most of the execution scenarios. Applications' performance with SimAS degraded in some instances due to the non-preemptive scheduling implementation. Even though the tech-

nique with the best performance is selected upon a new call to SimAS, the execution of already scheduled loop iterations can not be preempted to be resumed with the newly selected DLS. The results show that the overhead of SimAS is less than 0.5% of the application execution time (see Figures 12.11 and 12.12).

**Native performance with time-stepping applications under perturbations** To show the applicability of SimAS approach to scientific applications, time-stepping versions of PSIA and Mandelbrot are also executed under perturbations with and without SimAS. In time-stepping applications, i.e., PSIA_TS and Mandelbrot_TS, SimAS starts a new simulation at the beginning of each time step. WF is used as the default DLS technique in these experiments or the same DLS from the previous time-steps until the simulations are finished. SimAS selects the best performing DLS techniques based on the prediction from simulations for the current time-step. This represents another use-case of SimAS in time-stepping applications, which is frequently encountered in scientific applications. The results of the time-stepping applications are shown in Figures 12.13 and 12.14. Similar to the non-time-stepping versions, SimAS improved the performance of applications in most of the cases. We note that no single DLS technique always achieves the best performance. Therefore, a dynamic selection of the DLS technique according to the current perturbations in the system is needed. The SimAS overhead is, in general, below 0.5% of the execution time, except for PSIA_TS, which has the overhead of 2.7% at the most. This is due to the short execution time of the time-stepping version of the PSIA compared to the non-time-stepping version.

### 12.2.3 Discussion

Even though the applications considered are computationally-intensive and only communicate loop indices with the master, *perturbations in network latency had a significant impact on performance*. The implementation choice of the scheduling techniques, such as STATIC, implemented in a self-scheduling fashion, led to degrading its performance in scenarios with network perturbations. In most experiments, all the adaptive DLS techniques perform comparably. However, in certain instances, e.g., AWF-C and AF in Figure 12.12 in `lat-cm` and `lat-cs`, their performance was significantly poorer compared to other adaptive DLS techniques. This poor performance is due to the short execution time of the Mandelbrot application and the high variability of the loop iteration execution times, in addition to the added perturbations, which does not allow the core weights learned by these techniques to converge to the correct value.

Selecting the most performing DLS technique before execution might not deliver the best performance, as perturbations in the HPC system are unknown a priori. For

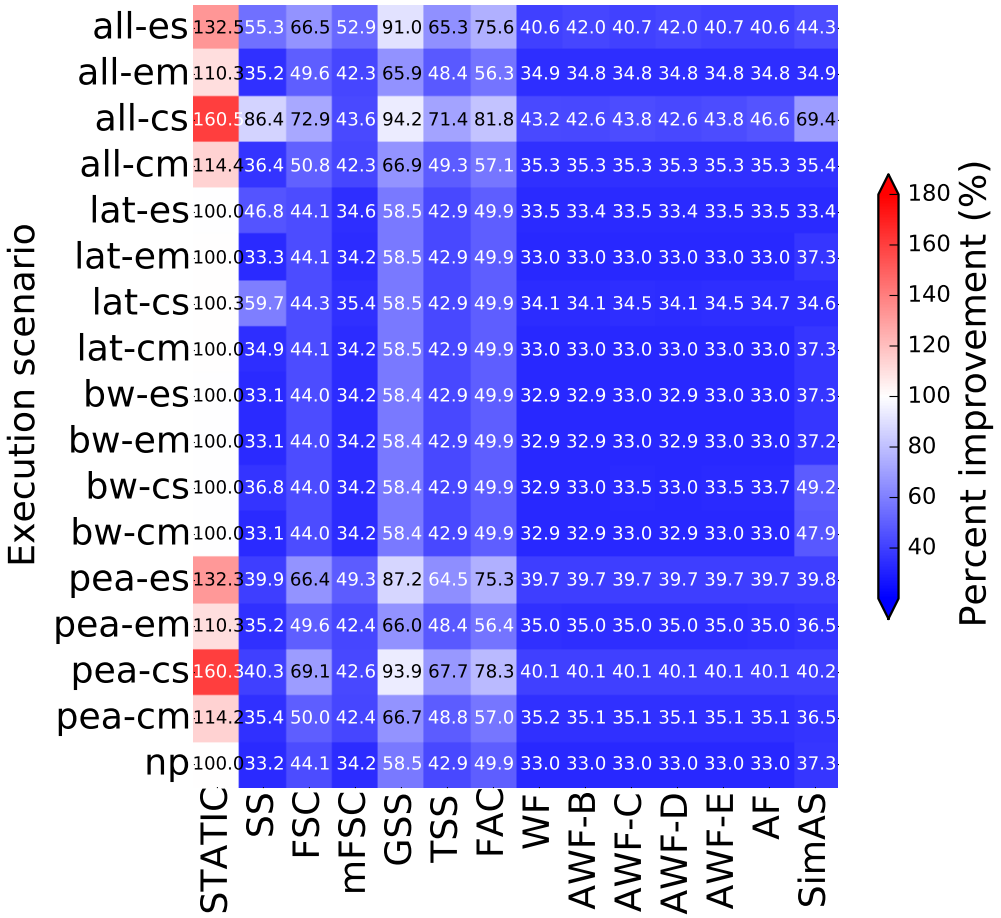
instance, the best DLS technique for Mandelbrot that could be identified before execution, i.e., in *np* execution scenario, is SS, which is outperformed by SimAS in *lat-cs* and *pea+lat-cs* in Figure 12.12. A similar change in the best DLS technique is observed in the results of Mandelbrot_TS in Figure 12.14. Since there is no high load imbalance in the PSIA or PSIA_TS, there is no high variation in the performance of different DLS techniques. *Since the best DLS technique can not be known before execution, SimAS improved the performance by dynamically selecting the DLS with the best performance based on the simulation predictions.*

In general, DLS techniques are designed to be efficient. However, efficiency prevents robustness due to the low tolerance of efficient techniques to uncertain events. Uncertainty is ineradicable, and it manifests in HPC systems as perturbations (see Chapter 3). This highlights the importance of the careful choice of DLS techniques for each application, system size, and execution scenario. Dynamic selection of DLS techniques ensures that each DLS technique is employed where it is the most efficient.

The SimAS approach can proactively select the best suited DLS before any perturbations manifest in the system, whenever perturbations can be predicted in advance. The SimAS leverages the use of already developed simulators, instead of needing the development of novel prediction techniques. The DLS selection decisions taken by SimAS can then be used to create a rule-based DLS selection mechanism for a combination of application, system, and execution scenarios, to improve application performance dynamically without the need of online simulation.

Running SimAS simulations and the dynamic selection of DLS techniques incurs overhead. However, this overhead has a limited effect on applications' performance. For example, the total time spent in `SimAS_setup` and `SimAS_update` functions is 3.49 seconds out of 1147.55 total application execution time for the PSIA on 128 cores in the `lat-cs` execution scenario. However, due to the non-preemptive property of the DLS, the execution of already scheduled chunks of loop iterations is not preempted to be resumed with the newly selected DLS. As shown in Figure 12.11(b), even though the SimAS selected DLS techniques with shorter execution times in the case of `lat-cs` with PSIA application on 128 cores, the execution time with SimAS was even longer than that of SS, which was not selected by the SimAS.

In time-stepping applications, the effect of frequent DLS technique switching and the non-preemption overhead is much less than the single-sweep applications. Therefore, the performance of time-stepping applications with SimAS under perturbations is better than the single-sweep versions of the same applications as can be seen in Figures 12.13 and 12.14. The preemption of scheduled (yet not executed) loop iterations may improve the performance while switching DLS techniques.

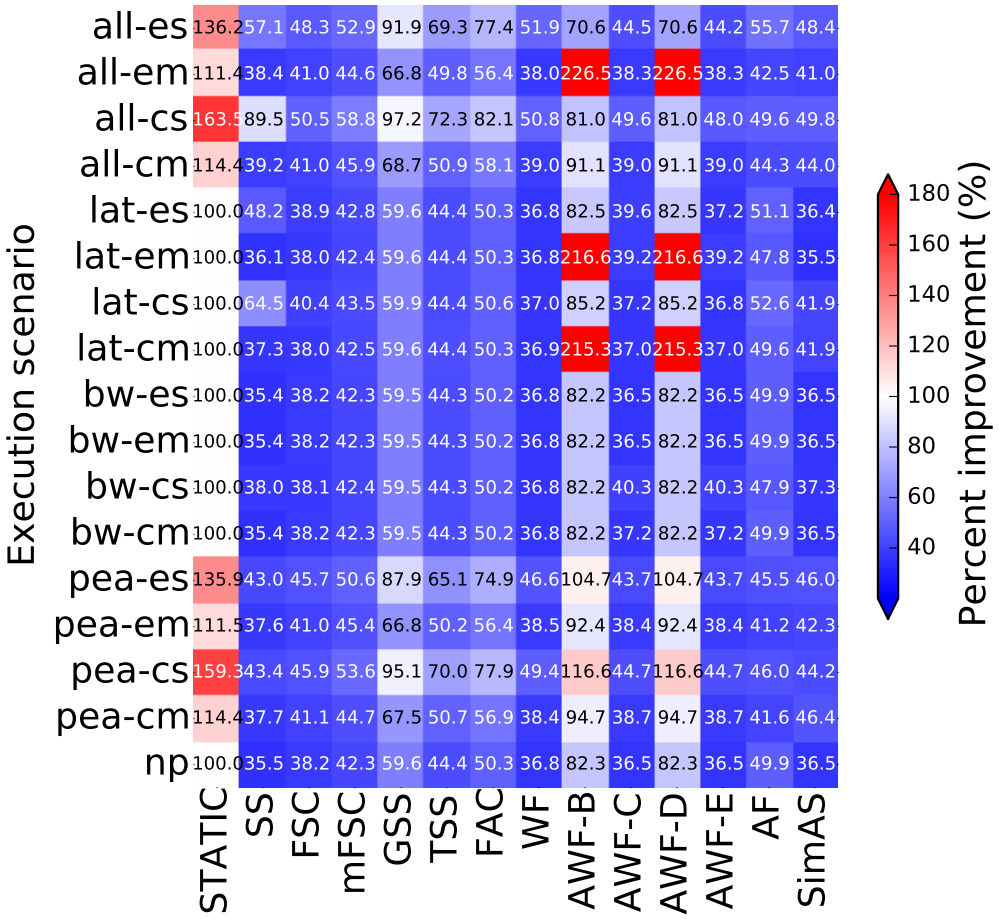



(a) PSIA simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 44.4% | 0.0% | 22.2% | 0.0% | 0.0% | 0.0% | 11.1% | 22.2% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 25.0% | 0.0% | 25.0% | 0.0% | 0.0% | 0.0% | 0.0% | 25.0% | 12.5% | 0.0% | 12.5% | 0.0% |
| **all-cs** | 0.0% | 14.3% | 7.1% | 14.3% | 0.0% | 0.0% | 0.0% | 42.9% | 21.4% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 37.5% | 0.0% | 12.5% | 0.0% | 0.0% | 0.0% | 12.5% | 37.5% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 0.0% | 0.0% | 28.6% | 0.0% | 0.0% | 0.0% | 28.6% | 28.6% | 0.0% | 0.0% | 0.0% | 14.3% |
| **lat-em** | 0.0% | 0.0% | 12.5% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 12.5% |
| **lat-cs** | 0.0% | 0.0% | 0.0% | 28.6% | 0.0% | 0.0% | 0.0% | 28.6% | 28.6% | 14.3% | 0.0% | 0.0% | 0.0% |
| **lat-cm** | 0.0% | 0.0% | 0.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% |
| **bw-es** | 0.0% | 0.0% | 0.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% |
| **bw-em** | 0.0% | 0.0% | 0.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% |
| **bw-cs** | 0.0% | 0.0% | 10.0% | 40.0% | 0.0% | 0.0% | 0.0% | 30.0% | 10.0% | 10.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 0.0% | 20.0% | 20.0% | 0.0% | 0.0% | 0.0% | 40.0% | 20.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-es** | 0.0% | 55.6% | 0.0% | 11.1% | 0.0% | 0.0% | 0.0% | 11.1% | 22.2% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 12.5% | 0.0% | 37.5% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 44.4% | 0.0% | 33.3% | 0.0% | 0.0% | 0.0% | 11.1% | 11.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 12.5% | 0.0% | 37.5% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **np** | 0.0% | 0.0% | 0.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.4 Simulative performance results of PSIA without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

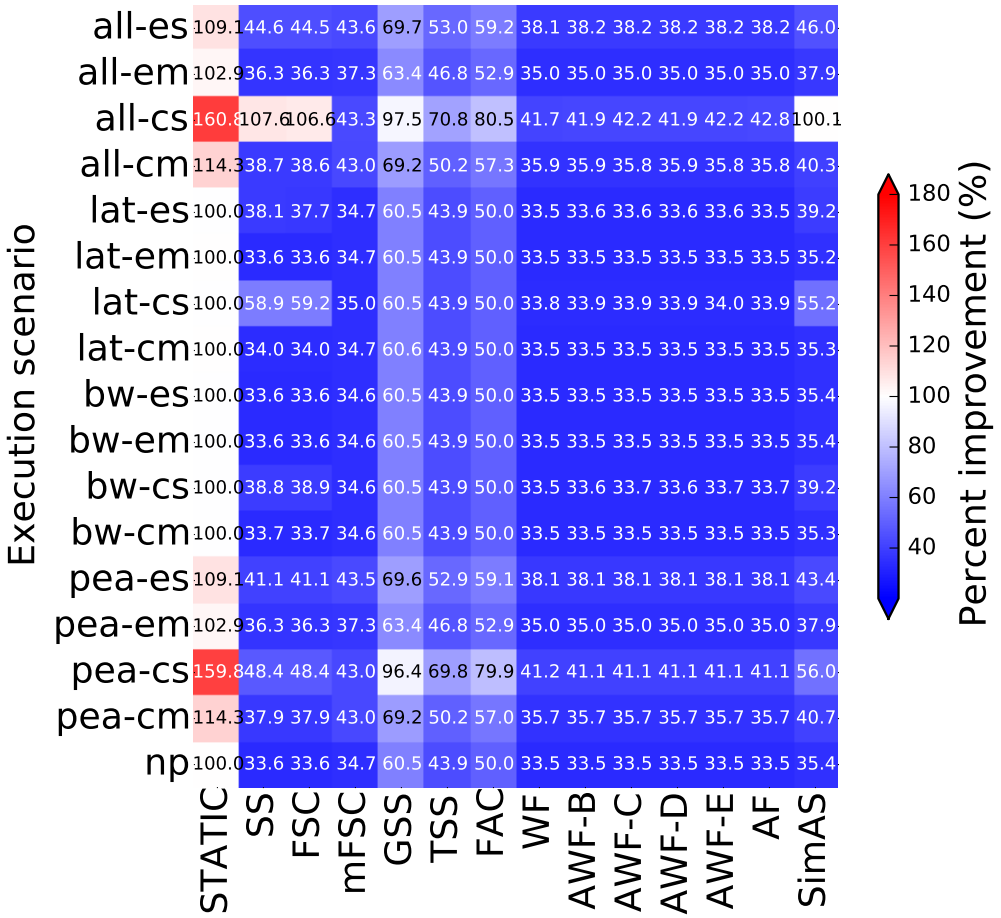



(a) Mandelbrot simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 11.1% | 55.6% | 0.0% | 0.0% | 0.0% | 0.0% | 11.1% | 22.2% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 12.5% | 37.5% | 37.5% | 0.0% | 0.0% | 0.0% | 12.5% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cs** | 0.0% | 10.0% | 40.0% | 20.0% | 0.0% | 0.0% | 0.0% | 20.0% | 10.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 25.0% | 25.0% | 12.5% | 0.0% | 0.0% | 0.0% | 12.5% | 25.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 14.3% | 28.6% | 14.3% | 0.0% | 0.0% | 0.0% | 14.3% | 28.6% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-em** | 0.0% | 28.6% | 42.9% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 28.6% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-cs** | 0.0% | 12.5% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 25.0% | 12.5% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-cm** | 0.0% | 0.0% | 50.0% | 0.0% | 0.0% | 0.0% | 0.0% | 25.0% | 12.5% | 0.0% | 0.0% | 12.5% | 0.0% |
| **bw-es** | 0.0% | 14.3% | 42.9% | 14.3% | 0.0% | 0.0% | 0.0% | 14.3% | 14.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-em** | 0.0% | 14.3% | 42.9% | 14.3% | 0.0% | 0.0% | 0.0% | 14.3% | 14.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cs** | 0.0% | 14.3% | 42.9% | 28.6% | 0.0% | 0.0% | 0.0% | 14.3% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 14.3% | 42.9% | 14.3% | 0.0% | 0.0% | 0.0% | 14.3% | 14.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-es** | 0.0% | 22.2% | 33.3% | 33.3% | 0.0% | 0.0% | 0.0% | 11.1% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 25.0% | 25.0% | 25.0% | 0.0% | 0.0% | 0.0% | 0.0% | 25.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 22.2% | 44.4% | 11.1% | 0.0% | 0.0% | 0.0% | 11.1% | 11.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 11.1% | 22.2% | 0.0% | 0.0% | 0.0% | 0.0% | 33.3% | 22.2% | 0.0% | 11.1% | 0.0% | 0.0% |
| **np** | 0.0% | 14.3% | 42.9% | 14.3% | 0.0% | 0.0% | 0.0% | 14.3% | 14.3% | 0.0% | 0.0% | 0.0% | 0.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.5 Simulative performance results of Mandelbrot without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

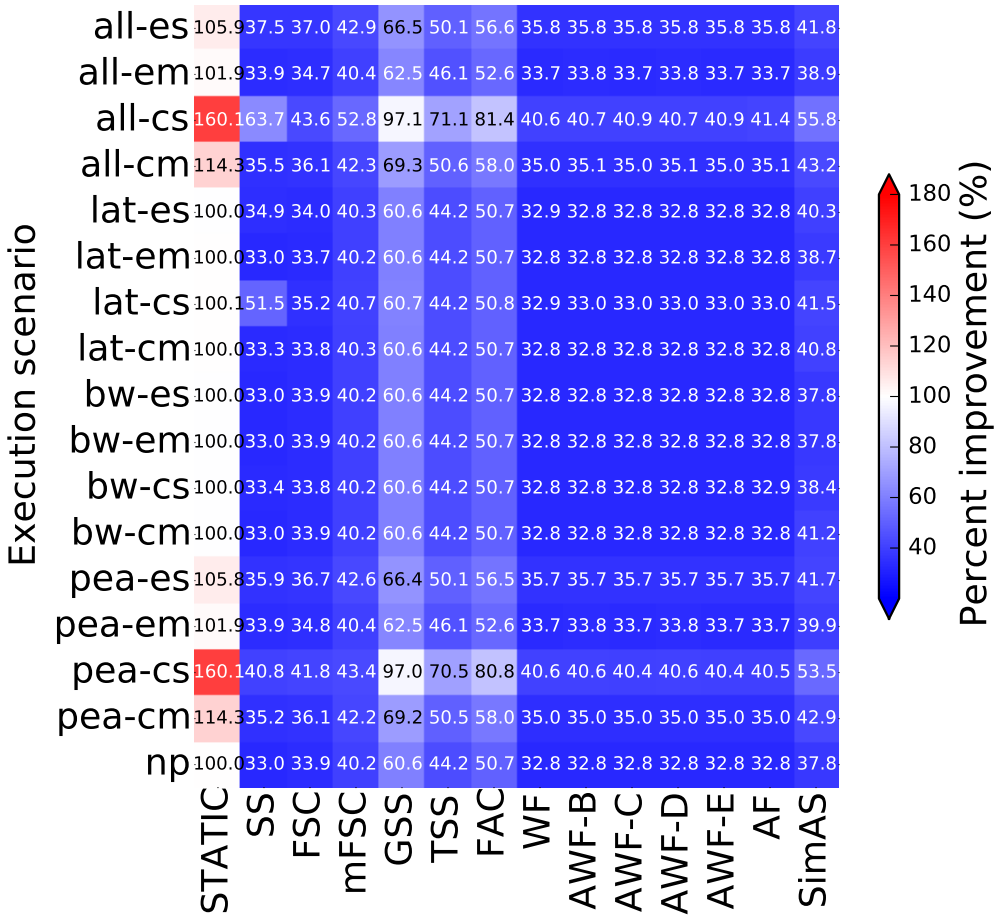



(a) Constant workload simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 36.4% | 54.5% | 3.0% | 0.0% | 0.0% | 0.0% | 3.0% | 0.0% | 3.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 46.4% | 39.3% | 3.6% | 0.0% | 0.0% | 0.0% | 10.7% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cs** | 0.0% | 28.8% | 57.5% | 2.7% | 0.0% | 0.0% | 0.0% | 9.6% | 1.4% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 58.6% | 24.1% | 6.9% | 0.0% | 0.0% | 0.0% | 10.3% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 42.9% | 42.9% | 7.1% | 0.0% | 0.0% | 0.0% | 7.1% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-em** | 0.0% | 38.5% | 38.5% | 11.5% | 0.0% | 0.0% | 0.0% | 7.7% | 0.0% | 0.0% | 0.0% | 0.0% | 3.8% |
| **lat-cs** | 0.0% | 53.7% | 31.7% | 4.9% | 0.0% | 0.0% | 0.0% | 7.3% | 0.0% | 0.0% | 0.0% | 0.0% | 2.4% |
| **lat-cm** | 0.0% | 46.2% | 30.8% | 7.7% | 0.0% | 0.0% | 0.0% | 11.5% | 0.0% | 0.0% | 0.0% | 3.8% | 0.0% |
| **bw-es** | 0.0% | 0.0% | 80.0% | 4.0% | 0.0% | 0.0% | 0.0% | 12.0% | 4.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-em** | 0.0% | 0.0% | 80.0% | 4.0% | 0.0% | 0.0% | 0.0% | 12.0% | 4.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cs** | 0.0% | 39.3% | 42.9% | 7.1% | 0.0% | 0.0% | 0.0% | 10.7% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 4.0% | 80.0% | 8.0% | 0.0% | 0.0% | 0.0% | 4.0% | 0.0% | 0.0% | 0.0% | 4.0% | 0.0% |
| **pea-es** | 0.0% | 29.0% | 51.6% | 0.0% | 0.0% | 0.0% | 0.0% | 12.9% | 3.2% | 3.2% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 77.8% | 11.1% | 3.7% | 0.0% | 0.0% | 0.0% | 7.4% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 35.0% | 40.0% | 0.0% | 0.0% | 0.0% | 0.0% | 20.0% | 2.5% | 2.5% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 72.4% | 6.9% | 0.0% | 0.0% | 0.0% | 0.0% | 10.3% | 3.4% | 6.9% | 0.0% | 0.0% | 0.0% |
| **np** | 0.0% | 0.0% | 80.0% | 4.0% | 0.0% | 0.0% | 0.0% | 12.0% | 4.0% | 0.0% | 0.0% | 0.0% | 0.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.6** **Simulative performance results of Constant synthetic workload without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

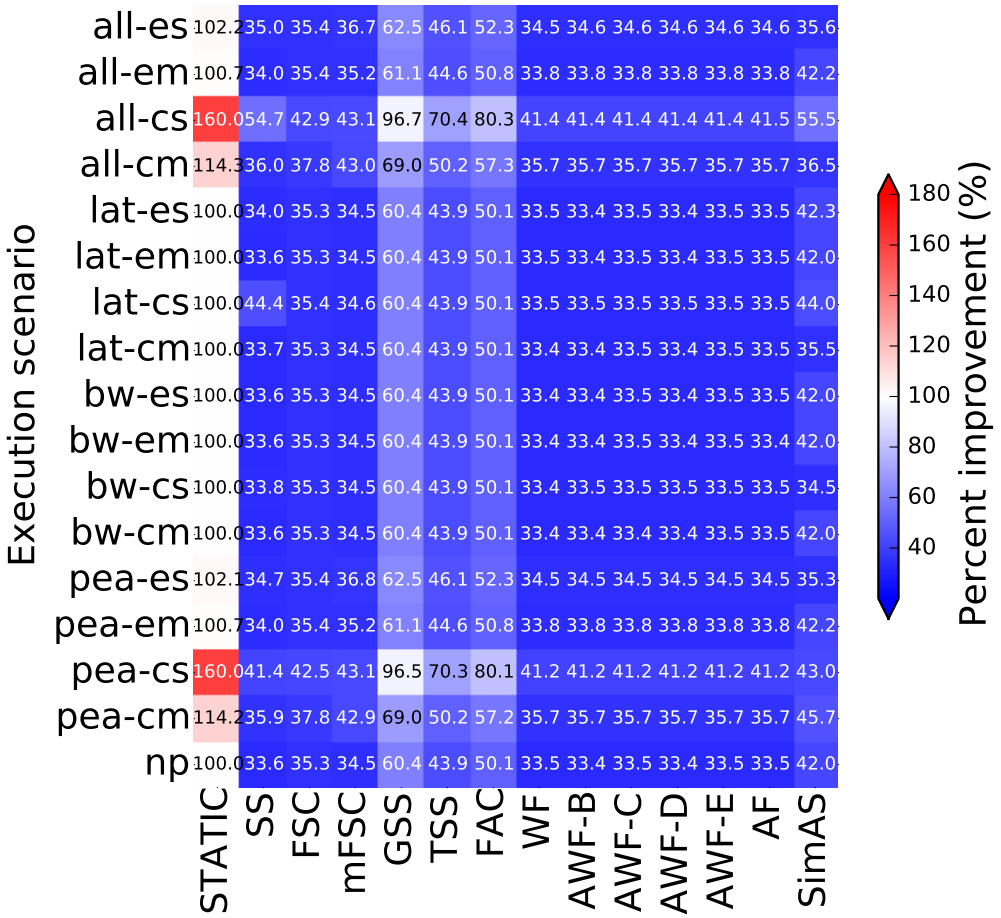



(a) Uniform workload simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 20.8% | 56.2% | 0.0% | 0.0% | 0.0% | 0.0% | 20.8% | 2.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 2.3% | 77.3% | 2.3% | 0.0% | 0.0% | 0.0% | 18.2% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cs** | 0.0% | 19.4% | 54.8% | 0.0% | 0.0% | 0.0% | 0.0% | 24.2% | 1.6% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 2.0% | 73.5% | 0.0% | 0.0% | 0.0% | 0.0% | 22.4% | 2.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 0.0% | 73.9% | 0.0% | 0.0% | 0.0% | 0.0% | 19.6% | 2.2% | 4.3% | 0.0% | 0.0% | 0.0% |
| **lat-em** | 0.0% | 4.5% | 72.7% | 0.0% | 0.0% | 0.0% | 0.0% | 18.2% | 4.5% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-cs** | 0.0% | 2.1% | 74.5% | 2.1% | 0.0% | 0.0% | 0.0% | 19.1% | 0.0% | 2.1% | 0.0% | 0.0% | 0.0% |
| **lat-cm** | 0.0% | 0.0% | 71.7% | 2.2% | 0.0% | 0.0% | 0.0% | 23.9% | 0.0% | 0.0% | 0.0% | 2.2% | 0.0% |
| **bw-es** | 0.0% | 2.3% | 76.7% | 2.3% | 0.0% | 0.0% | 0.0% | 18.6% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-em** | 0.0% | 2.3% | 76.7% | 2.3% | 0.0% | 0.0% | 0.0% | 18.6% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cs** | 0.0% | 4.7% | 74.4% | 2.3% | 0.0% | 0.0% | 0.0% | 16.3% | 2.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 0.0% | 70.2% | 2.1% | 0.0% | 0.0% | 0.0% | 25.5% | 0.0% | 2.1% | 0.0% | 0.0% | 0.0% |
| **pea-es** | 0.0% | 14.6% | 64.6% | 2.1% | 0.0% | 0.0% | 0.0% | 16.7% | 2.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 4.4% | 73.3% | 0.0% | 0.0% | 0.0% | 0.0% | 17.8% | 4.4% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 16.7% | 56.7% | 0.0% | 0.0% | 0.0% | 0.0% | 25.0% | 1.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 4.1% | 75.5% | 0.0% | 0.0% | 0.0% | 0.0% | 18.4% | 2.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **np** | 0.0% | 2.3% | 76.7% | 2.3% | 0.0% | 0.0% | 0.0% | 18.6% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.7 Simulative performance results of Uniform synthetic workload without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

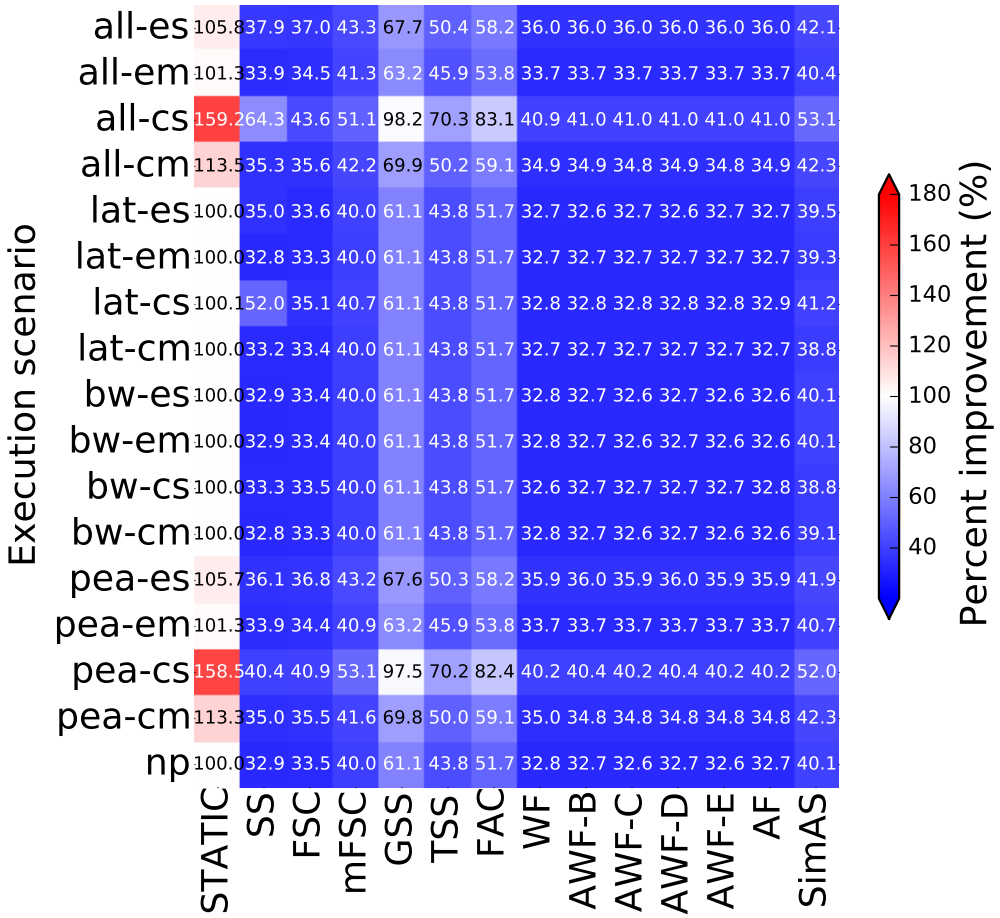



(a) Normal workload simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 92.6% | 5.6% | 0.0% | 0.0% | 0.0% | 0.0% | 1.9% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 75.6% | 3.1% | 0.0% | 0.0% | 0.0% | 0.0% | 20.5% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cs** | 0.0% | 93.1% | 1.7% | 0.0% | 0.0% | 0.0% | 0.0% | 4.6% | 0.6% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 91.3% | 1.7% | 0.0% | 0.0% | 0.0% | 0.0% | 6.1% | 0.9% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 75.6% | 3.1% | 0.0% | 0.0% | 0.0% | 0.0% | 20.5% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-em** | 0.0% | 75.4% | 3.2% | 0.0% | 0.0% | 0.0% | 0.0% | 20.6% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-cs** | 0.0% | 92.0% | 2.9% | 0.0% | 0.0% | 0.0% | 0.0% | 4.3% | 0.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-cm** | 0.0% | 87.2% | 5.5% | 0.0% | 0.0% | 0.0% | 0.0% | 6.4% | 0.0% | 0.0% | 0.0% | 0.9% | 0.0% |
| **bw-es** | 0.0% | 75.4% | 3.2% | 0.0% | 0.0% | 0.0% | 0.0% | 20.6% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-em** | 0.0% | 75.4% | 3.2% | 0.0% | 0.0% | 0.0% | 0.0% | 20.6% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cs** | 0.0% | 89.6% | 5.7% | 0.9% | 0.0% | 0.0% | 0.0% | 3.8% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 75.4% | 3.2% | 0.0% | 0.0% | 0.0% | 0.0% | 20.6% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-es** | 0.0% | 89.1% | 4.5% | 0.0% | 0.0% | 0.0% | 0.0% | 5.5% | 0.9% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 75.6% | 3.1% | 0.0% | 0.0% | 0.0% | 0.0% | 20.5% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 91.8% | 1.5% | 0.0% | 0.0% | 0.0% | 0.0% | 6.0% | 0.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 73.7% | 3.6% | 0.0% | 0.0% | 0.0% | 0.0% | 21.9% | 0.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **np** | 0.0% | 75.4% | 3.2% | 0.0% | 0.0% | 0.0% | 0.0% | 20.6% | 0.8% | 0.0% | 0.0% | 0.0% | 0.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.8 Simulative performance results of Normal synthetic workload without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

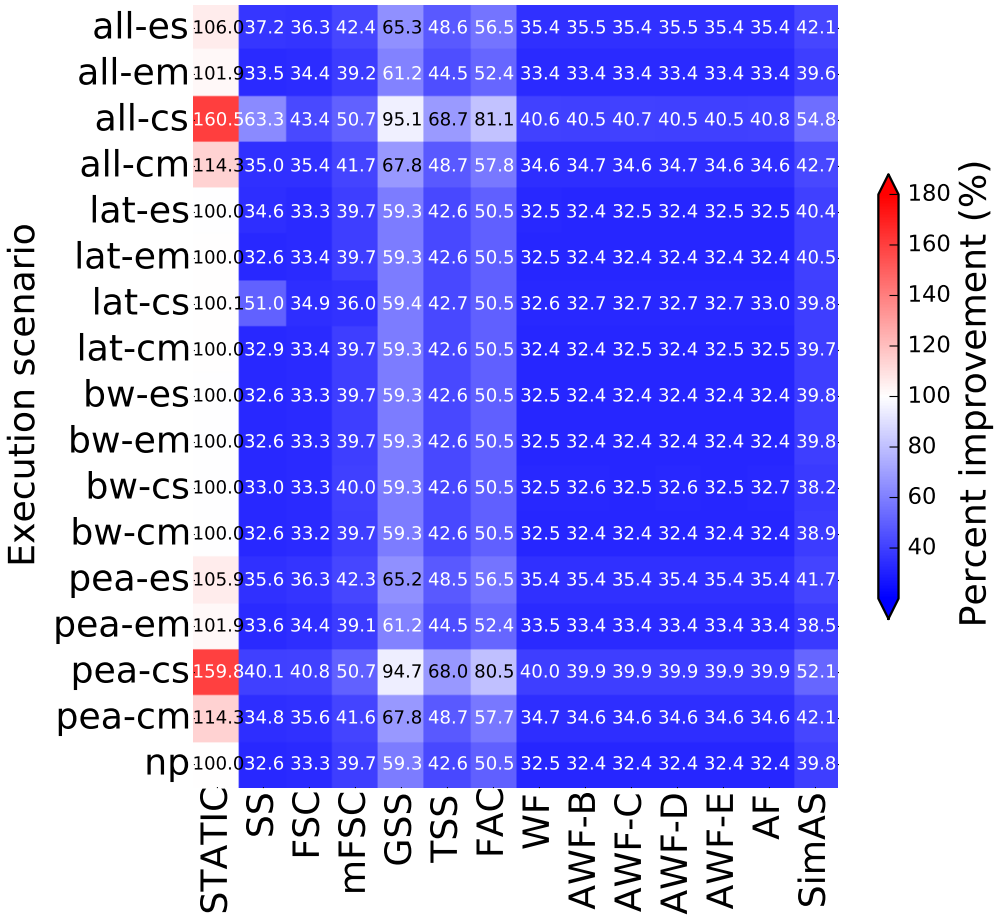



(a) Exponential workload simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 4.7% | 72.1% | 0.0% | 0.0% | 0.0% | 0.0% | 18.6% | 4.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 0.0% | 73.2% | 2.4% | 0.0% | 0.0% | 0.0% | 22.0% | 2.4% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cs** | 0.0% | 9.3% | 66.7% | 1.9% | 0.0% | 0.0% | 0.0% | 22.2% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 4.7% | 72.1% | 0.0% | 0.0% | 0.0% | 0.0% | 20.9% | 2.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 5.0% | 70.0% | 2.5% | 0.0% | 0.0% | 0.0% | 20.0% | 2.5% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-em** | 0.0% | 2.5% | 72.5% | 0.0% | 0.0% | 0.0% | 0.0% | 20.0% | 2.5% | 0.0% | 0.0% | 2.5% | 0.0% |
| **lat-cs** | 0.0% | 0.0% | 73.8% | 0.0% | 0.0% | 0.0% | 0.0% | 21.4% | 4.8% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-cm** | 0.0% | 0.0% | 71.8% | 2.6% | 0.0% | 0.0% | 0.0% | 23.1% | 2.6% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-es** | 0.0% | 7.3% | 68.3% | 0.0% | 0.0% | 0.0% | 0.0% | 19.5% | 2.4% | 0.0% | 0.0% | 2.4% | 0.0% |
| **bw-em** | 0.0% | 7.3% | 68.3% | 0.0% | 0.0% | 0.0% | 0.0% | 19.5% | 2.4% | 0.0% | 0.0% | 2.4% | 0.0% |
| **bw-cs** | 0.0% | 5.0% | 72.5% | 2.5% | 0.0% | 0.0% | 0.0% | 20.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 0.0% | 72.5% | 0.0% | 0.0% | 0.0% | 0.0% | 22.5% | 2.5% | 0.0% | 0.0% | 0.0% | 2.5% |
| **pea-es** | 0.0% | 9.3% | 72.1% | 2.3% | 0.0% | 0.0% | 0.0% | 16.3% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 9.5% | 69.0% | 0.0% | 0.0% | 0.0% | 0.0% | 19.0% | 2.4% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 5.8% | 67.3% | 0.0% | 0.0% | 0.0% | 0.0% | 25.0% | 1.9% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 7.0% | 72.1% | 2.3% | 0.0% | 0.0% | 0.0% | 18.6% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **np** | 0.0% | 7.3% | 68.3% | 0.0% | 0.0% | 0.0% | 0.0% | 19.5% | 2.4% | 0.0% | 0.0% | 2.4% | 0.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.9 Simulative performance results of Exponential synthetic workload without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

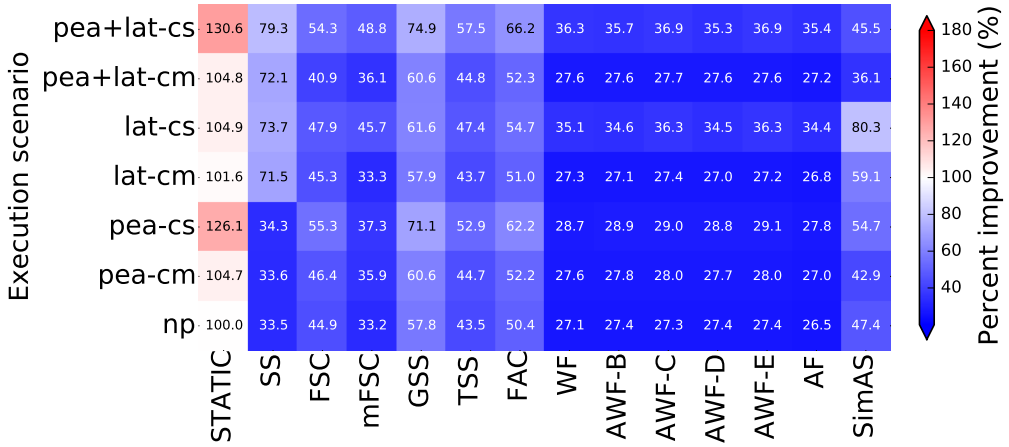



(a) Gamma workload simulative performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **all-es** | 0.0% | 21.3% | 57.4% | 0.0% | 0.0% | 0.0% | 0.0% | 19.1% | 2.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-em** | 0.0% | 4.5% | 75.0% | 4.5% | 0.0% | 0.0% | 0.0% | 15.9% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cs** | 0.0% | 18.3% | 55.0% | 1.7% | 0.0% | 0.0% | 0.0% | 23.3% | 1.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **all-cm** | 0.0% | 6.2% | 72.9% | 0.0% | 0.0% | 0.0% | 0.0% | 18.8% | 2.1% | 0.0% | 0.0% | 0.0% | 0.0% |
| **lat-es** | 0.0% | 0.0% | 71.1% | 2.2% | 0.0% | 0.0% | 0.0% | 24.4% | 0.0% | 2.2% | 0.0% | 0.0% | 0.0% |
| **lat-em** | 0.0% | 4.4% | 71.1% | 2.2% | 0.0% | 0.0% | 0.0% | 20.0% | 0.0% | 0.0% | 2.2% | 0.0% | 0.0% |
| **lat-cs** | 0.0% | 0.0% | 75.0% | 2.3% | 0.0% | 0.0% | 0.0% | 18.2% | 2.3% | 0.0% | 2.3% | 0.0% | 0.0% |
| **lat-cm** | 0.0% | 0.0% | 75.0% | 0.0% | 0.0% | 0.0% | 0.0% | 20.5% | 2.3% | 0.0% | 0.0% | 2.3% | 0.0% |
| **bw-es** | 0.0% | 6.8% | 72.7% | 0.0% | 0.0% | 0.0% | 0.0% | 18.2% | 2.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-em** | 0.0% | 6.8% | 72.7% | 0.0% | 0.0% | 0.0% | 0.0% | 18.2% | 2.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cs** | 0.0% | 2.4% | 78.6% | 2.4% | 0.0% | 0.0% | 0.0% | 16.7% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **bw-cm** | 0.0% | 4.5% | 72.7% | 2.3% | 0.0% | 0.0% | 0.0% | 20.5% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-es** | 0.0% | 12.8% | 61.7% | 0.0% | 0.0% | 0.0% | 0.0% | 19.1% | 4.3% | 2.1% | 0.0% | 0.0% | 0.0% |
| **pea-em** | 0.0% | 2.3% | 74.4% | 2.3% | 0.0% | 0.0% | 0.0% | 18.6% | 2.3% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cs** | 0.0% | 10.3% | 62.1% | 0.0% | 0.0% | 0.0% | 0.0% | 25.9% | 1.7% | 0.0% | 0.0% | 0.0% | 0.0% |
| **pea-cm** | 0.0% | 4.3% | 72.3% | 4.3% | 0.0% | 0.0% | 0.0% | 19.1% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| **np** | 0.0% | 6.8% | 72.7% | 0.0% | 0.0% | 0.0% | 0.0% | 18.2% | 2.3% | 0.0% | 0.0% | 0.0% | 0.0% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.10 Simulative performance results of Gamma synthetic workload without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS during execution.

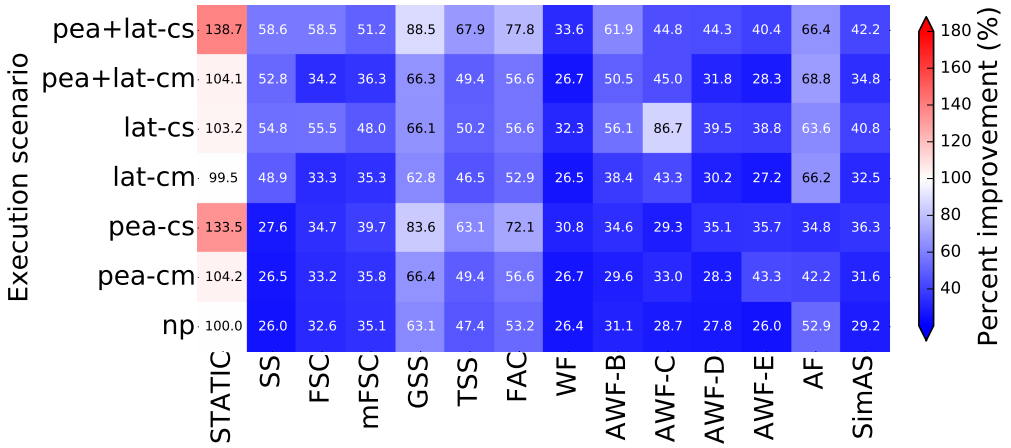



(a) PSIA native performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF | SimAS overhead |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pea+lat-cs | 0.0% | 0.0% | 55.0% | 12.6% | 0.0% | 0.0% | 0.0% | 32.5% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.4% |
| pea+lat-cm | 0.0% | 0.0% | 32.3% | 16.7% | 0.0% | 0.0% | 0.0% | 31.2% | 14.6% | 2.1% | 3.1% | 0.0% | 0.0% | 0.7% |
| lat-cs | 0.0% | 0.0% | 5.1% | 11.8% | 0.0% | 0.0% | 0.0% | 19.9% | 42.6% | 1.5% | 18.4% | 0.7% | 0.0% | 0.4% |
| lat-cm | 0.0% | 0.0% | 1.7% | 7.6% | 0.0% | 0.0% | 0.0% | 23.5% | 35.3% | 12.6% | 13.4% | 3.4% | 2.5% | 0.5% |
| pea-cs | 0.0% | 0.0% | 4.1% | 18.2% | 0.0% | 0.0% | 0.0% | 13.2% | 33.9% | 12.4% | 8.3% | 5.0% | 5.0% | 0.6% |
| pea-cm | 0.0% | 0.0% | 0.8% | 16.7% | 0.0% | 0.0% | 0.0% | 24.2% | 27.5% | 18.3% | 5.0% | 4.2% | 3.3% | 0.6% |
| np | 0.0% | 0.0% | 4.5% | 17.0% | 0.0% | 0.0% | 0.0% | 18.8% | 24.1% | 13.4% | 7.1% | 11.6% | 3.6% | 0.5% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.11 Native performance results of PSIA without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS and the percent of execution time spent in SimAS calls.

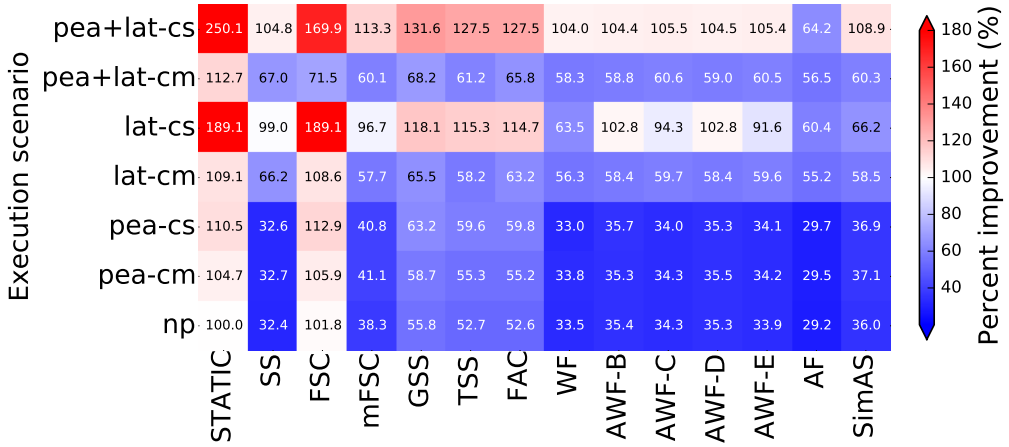



(a) Mandelbrot native performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF | SimAS overhead |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **pea+lat-cs** | 0.0% | 0.0% | 11.8% | 36.5% | 0.0% | 0.0% | 0.0% | 38.8% | 7.1% | 0.0% | 5.9% | 0.0% | 0.0% | 0.2% |
| **pea+lat-cm** | 0.0% | 0.0% | 22.5% | 21.3% | 0.0% | 0.0% | 0.0% | 23.6% | 11.2% | 0.0% | 7.9% | 0.0% | 13.5% | 0.3% |
| **lat-cs** | 0.0% | 0.0% | 1.1% | 12.5% | 0.0% | 0.0% | 0.0% | 40.9% | 30.7% | 0.0% | 14.8% | 0.0% | 0.0% | 0.3% |
| **lat-cm** | 0.0% | 0.0% | 34.4% | 19.8% | 0.0% | 0.0% | 0.0% | 20.8% | 3.1% | 1.0% | 7.3% | 0.0% | 13.5% | 0.3% |
| **pea-cs** | 0.0% | 0.0% | 33.3% | 13.3% | 0.0% | 0.0% | 0.0% | 23.3% | 12.2% | 3.3% | 3.3% | 2.2% | 8.9% | 0.3% |
| **pea-cm** | 0.0% | 0.0% | 32.5% | 16.9% | 0.0% | 0.0% | 0.0% | 28.9% | 9.6% | 3.6% | 0.0% | 1.2% | 7.2% | 0.3% |
| **np** | 0.0% | 0.0% | 35.6% | 19.5% | 0.0% | 0.0% | 0.0% | 23.0% | 4.6% | 0.0% | 8.0% | 0.0% | 9.2% | 0.4% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.12 Native performance results of Mandelbrot without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. Each table shows the DLS techniques dynamically selected by SimAS and the percent of execution time spent in SimAS calls.

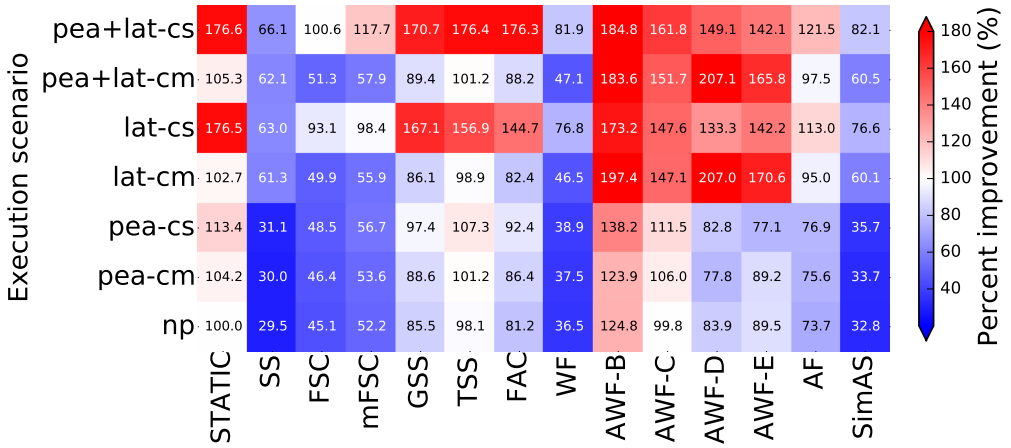



(a) PSIA_TS native performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF | SimAS overhead |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pea+lat-cs | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 100.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 1.7% |
| pea+lat-cm | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 86.2% | 13.8% | 0.0% | 0.0% | 0.0% | 0.0% | 2.0% |
| lat-cs | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 100.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 1.5% |
| lat-cm | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 100.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 1.7% |
| pea-cs | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 94.3% | 5.7% | 0.0% | 0.0% | 0.0% | 0.0% | 2.7% |
| pea-cm | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 93.5% | 6.5% | 0.0% | 0.0% | 0.0% | 0.0% | 2.7% |
| np | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 96.2% | 3.8% | 0.0% | 0.0% | 0.0% | 0.0% | 2.7% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.13 Native performance results of PSIA_TS without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. Each table shows the DLS techniques dynamically selected by SimAS and the percent of execution time spent in SimAS calls.




(a) Mandelbrot_TS native performance on 128 cores

| | STATIC | SS | FSC | mFSC | GSS | TSS | FAC | WF | AWF-B | AWF-C | AWF-D | AWF-E | AF | SimAS overhead |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pea+lat-cs | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 100.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.3% |
| pea+lat-cm | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.3% |
| lat-cs | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 100.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.2% |
| lat-cm | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.2% |
| pea-cs | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.5% |
| pea-cm | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.5% |
| np | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.0% | 0.0% | 0.0% | 0.0% | 50.0% | 0.4% |

(b) Percentage of counts DLS techniques are selected by SimAS

**Figure 12.14 Native performance results of Mandelbrot_TS without (denoted with np) and with (the rest) perturbations using SimAS and other thirteen loop scheduling techniques on miniHPC.** Percent performance improvement normalized to STATIC in np scenario (baseline case without any perturbations and baseline load balancing method). White, red, and blue denote baseline (= 100%), degraded (> 100%), and improved performance (< 100%), respectively. The table shows the DLS techniques dynamically selected by SimAS and the percent of execution time spent in SimAS calls.

# 13

# Robust DLS: Robustness Against Nonfatal and Fatal Perturbations

Parallel and independent loop iterations represent independent tasks, where a task denotes a single loop iteration. As shown in the previous chapters (see Chapter 2 and 3), load imbalance and perturbations (fatal and nonfatal) are two major challenges that degrade applications' performance on HPC systems. As HPC systems become larger to accommodate the computational needs of scientific applications, they become more vulnerable to faults and perturbations. The observed failure rate grows proportionally to the number of processor sockets in the system [SG07] with measured MTBF of $1-7$ days [BBC+; Ni16; DGG+19]. Extrapolating the current failure rates to Exascale systems results in an MTBF of 24 minutes, and if the resiliency of the components is assumed to be improved by a factor of 10, this results in a failure every 4 hours [SWA+14; BBC+].

Apart from failures, perturbations in the availability of the processing elements (PEs), network bandwidth, or network latency also degrade application performance. These perturbations occur due to operating system interference, transient malfunctions, or other applications sharing resources, such as the interconnection network [GSG+16; AE18; VLWB+19].

In this chapter, we answer the question of *"How to tolerate (fatal and nonfatal) perturbations during execution and maintain a load balanced execution on HPC systems?"*. We introduce the *robust Dynamic Load Balancing* (rDLB) approach for the robust self-scheduling of independent tasks under fail-stop failures or system performance variability (see Figure 13.1). rDLB proactively reschedules already *scheduled but not finished* tasks rather than reactively re-execute failed tasks. It does not require any failure or perturbation detection or measurement mechanism. rDLB extends the MPI-based *DLS4LB* library to enable robust scheduling in the presence of failures and perturbations on HPC systems. The FePIA procedure [AMS+04] (see Section 3.2) is used to evaluate the robust-

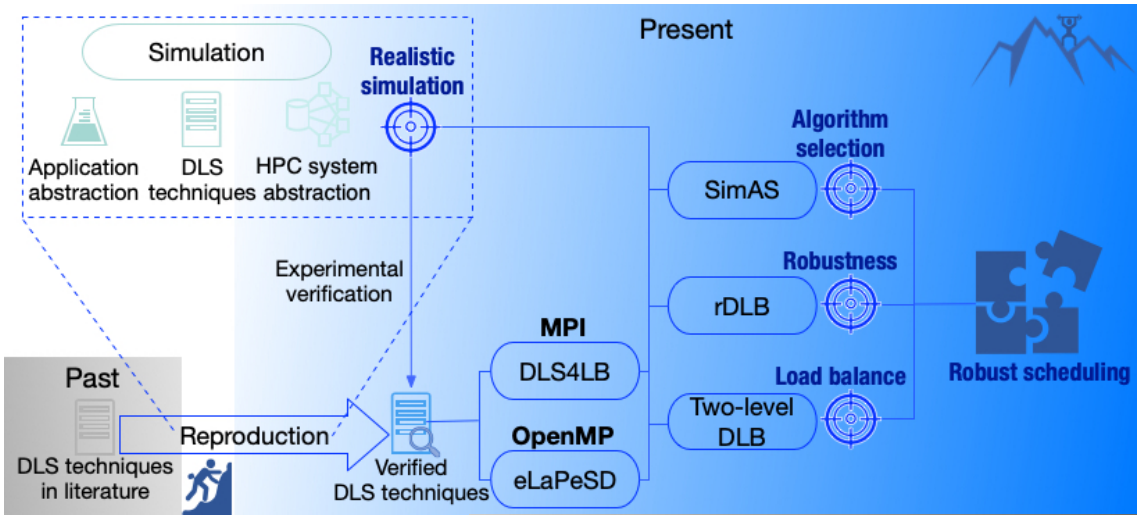



**Figure 13.1 Illustration of the focus and progress in this chapter as part of the overall approach.** rDLB is devised for the robust scheduling under fatal and nonfatal perturbations. rDLB, together with SimAS and the two-level DLB, achieve the robust scheduling needed for scientific applications executed on modern and future HPC systems.

ness of the proposed approach during the execution of two computationally-intensive scientific applications in different failure and perturbation scenarios.

## 13.1 rDLB: robust Dynamic Load Balancing

rDLB is a robust load balancing approach against fatal fail-stop failures (of PEs, compute nodes, or network elements that render PEs unreachable), as well as nonfatal perturbations in the PEs processing speed or the interconnection network. Using rDLB, each task is flagged as `Unscheduled`, `Scheduled`, or `Finished`. At the beginning of the execution, all tasks are `Unscheduled`. Tasks are self-scheduled via DLS techniques to free and requesting PEs, and their flags are changed from `Unscheduled` to `Scheduled`. In non-robust DLS execution, the scheduling operations end when all tasks are scheduled to PEs. In the proposed approach, scheduling, however, continues after all tasks are scheduled , to *reschedule scheduled but unfinished* tasks. The *key* idea of the proposed approach is to *leverage the idle time of PEs at the end of the execution to achieve robustness.* The execution completes when all tasks are executed and have their flags set to `Finished`.

Figure 13.2 illustrates an execution with and without the proposed robust DLB approach in the presence of a fail-stop failure. In the case of a PE failure, task `T4` will never be completed because it is already scheduled on processor $P3$, which subsequently failed, and the execution would wait indefinitely for $P3$ to finish task `T4` to complete the execu-

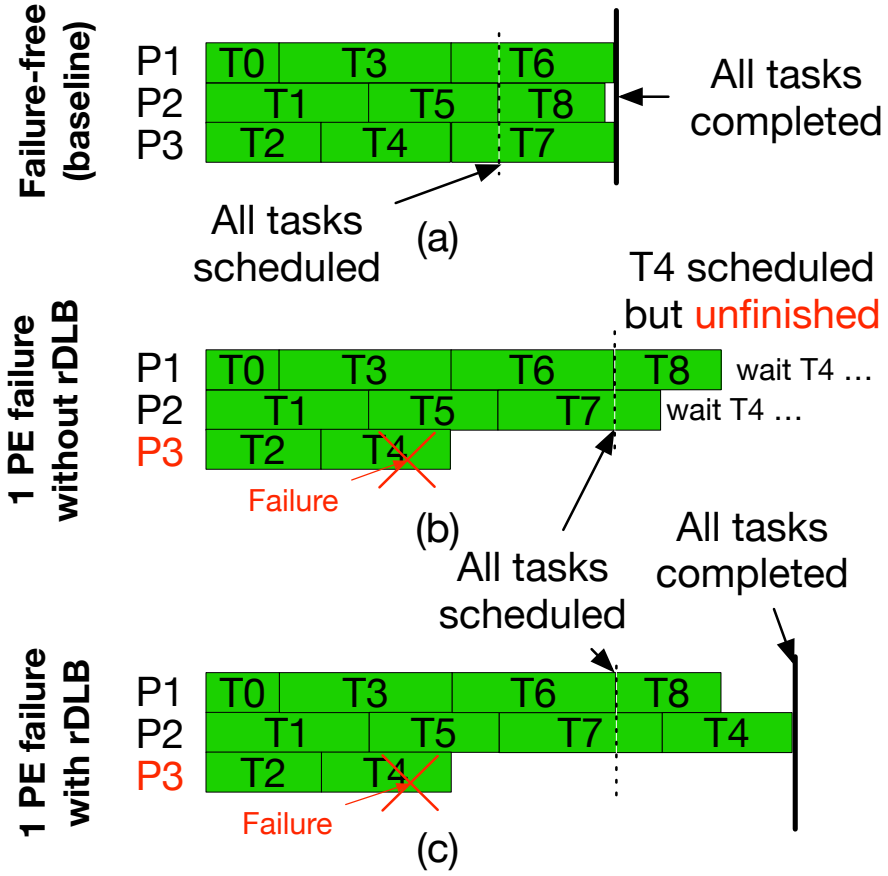



**Figure 13.2 Conceptual illustration of the execution of 9 tasks on 3 PEs *without* (a, b) and *with* (c) the proposed robust rDLB approach in the presence of failures (b, c).** Sub-figures show (a) the execution with the SS technique in the failure-free case without rDLB, (b) the execution with SS in the case of a single failure without rDLB, and (c) the execution with SS and rDLB in the case of a single failure [MCC19].

tion of the entire application. With rDLB, after all tasks are scheduled, and $P2$ becomes available and requests work, the first *scheduled but unfinished* task `T4` is assigned to it, and the execution completes as soon as all tasks are `Finished`.

Similarly, in the case of severe perturbations, illustrated in Figure 13.3, tasks assigned to $P2$, the perturbed PE, take much longer to complete. With rDLB approach, task `T7` is rescheduled on $P3$ and the execution completes earlier than that without rDLB. The advantage of the proposed rDLB approach is that it acts *proactively* and does not depend on any failure or perturbation mechanism. Rescheduling of unfinished tasks does not entirely add to the execution time, as it overlaps with the idle time when all tasks are scheduled and PEs are waiting for the completion of all tasks. The execution terminates as soon as either the original or the rescheduled tasks complete.

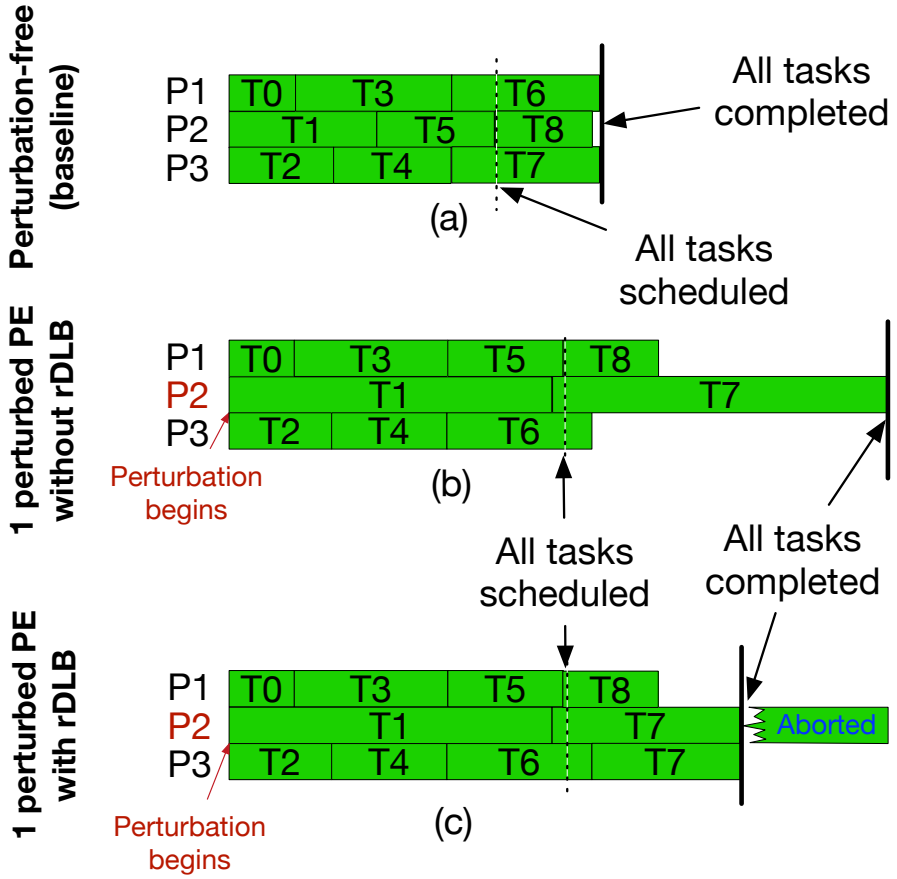



**Figure 13.3 Conceptual illustration of the execution of 9 tasks on 3 PEs *without* and *with* the proposed robust rDLB approach in the presence of perturbations.** Sub-figures show (a) the execution with SS in the perturbation-free case, (b) the execution with SS in the case of severe perturbations in *P2* without rDLB, and (c) the execution with SS and rDLB in the case of severe perturbations in *P2* [MCC19].

### 13.1.1 Analytical Modeling

From a theoretical point of view, it is possible to model the overhead of rDLB approach, due to re-execution or re-scheduling of tasks because of fatal and nonfatal perturbations.

Let $P$ denote the number of PEs and $N$ the total number of tasks. To simplify the model, we assume that all tasks are equal in size and are equally distributed among the PEs (which tends to be the case when $N >> P$). Since there are $n = \frac{N}{P}$ tasks per PE, the time needed to execute the entire application in a perturbations-free scenario is: $T = n \cdot t$, where $t$ is the time needed to execute one task.

Let $p_F^T$ denote the probability of having at least one PE failure during the execution. With probability $\left(1 - p_F^T\right)$, there is no failure and the application takes time $T$ to complete the execution. With probability $p_F^T$, there is a failure and we need to account for the extra time needed to execute the remaining tasks of the faulty processor. With probability $\frac{1}{n}$,

the error occurred on task $i$ and we need to execute the remaining $n - i$ tasks on the remaining $P - 1$ PEs. Accounting for all cases, we can write:

$$\begin{aligned}\mathbb{E}_T &= \left(1 - p_F^T\right) \cdot T + p_F^T \left(T + \sum_{i=0}^{n-1} \frac{1}{n} \cdot \frac{n-i}{P-1} \cdot t\right) \\ &= T + p_F^T \frac{t}{2} \frac{n+1}{P-1}.\end{aligned}$$

Assuming exponentially distributed failures, we can write $p_F^T = (1 - e^{-\lambda T})$, and since $T >> \lambda$, we can use the Taylor Series to approximate $p_F^T$ up to first-order terms and we obtain:

$$\mathbb{E}_T = T + \lambda T \frac{t}{2} \frac{n+1}{q-1} + O(\lambda^2 T^2).$$

Finally, we can compute the overhead:

$$\mathbb{H}_T = 1 - \frac{\mathbb{E}_T}{T} = \frac{\lambda t}{2} \frac{n+1}{P-1} + O(\lambda^2 T^2) = O\left(\lambda \frac{tn}{P}\right).$$

Since $T = n \cdot t$, we can also write:

$$\mathbb{H}_T = O\left(\lambda \frac{T}{P}\right),$$

which shows that the overhead of the rDLB approach decreases linearly with the number of PEs. Indeed, increasing the number of PEs helps mitigating both the number of tasks that need to be re-executed in the case of a failure and the number of tasks that are to be re-executed on the remaining PEs.

### 13.1.2 Implementation Details

The proposed rDLB approach is generic and can be used with any application that employs self-scheduling. rDLB is used to extend the *DLS4LB* scheduling library, thereby enhancing it with robustness features. The *DLS4LB* implements 13 DLS techniques (nonadaptive and adaptive) and employs a master-worker execution model using MPI two-sided communications (see Section 9.2).

Algorithm 13.1 shows the use of *DLS4LB* for robust scheduling. Lines in green highlight the required code lines to use the *DLS4LB*. Lines in blue highlight the code changes required to use rDLB in conjunction with *DLS4LB*. The MPI error handler is changed to `MPI_ERRORS_RETURN`, which reports MPI errors, rather than considering them fatal and ends the program immediately on MPI errors (default behavior). Workers ask the master for work whenever they become free. Upon completing a chunk of tasks, a worker sends the results to the master and asks for more work. Upon receiving the

**Algorithm 13.1 Use of rDLB in conjunction with *DLS4LB***

```
#include <mpi.h>
#include "DLS4LB.h"
MPI_Init(&argc, &argv)
MPI_Comm_size(MPI_COMM_WORLD, &P)
MPI_Comm_rank(MPI_COMM_WORLD, &myid)
/* Initialization                                                    */
1  rDLB_enabled = True
2  MPI_Comm_set_errhandler(MPI_COMM_WORLD, \
   MPI_ERRORS_RETURN)
3  DLS_setup(MPI_COMM_WORLD, DLS_info, rDLB_enabled)
4  DLS_startLoop (DLS_info, N, DLS_method, results_data)
5  while Not DLS_terminated do
6      DLS_startChunk(DLS_info, assigned_iters, size)
7      data = malloc(size)
       /* Main application loop                                     */
8      Compute_tasks(assigned_iters, size, data)
9      DLS_endChunk(DLS_info, data)
10     free(data)
11 DLS_endLoop(DLS_info)
   /* Save results                                                  */
12 ...
   /* Immediately end program, kill all processes                   */
13 MPI_Abort(MPI_COMM_WORLD, -1)
```

results of a chunk from a worker, the master marks the tasks previously assigned to the requesting worker as `Finished` and assigns the worker a new chunk. This operation is continued until all tasks are scheduled.

The *DLS4LB* is adjusted such that scheduling operations are continued until all tasks are finished. Therefore, the master continues assigning `Scheduled` tasks to requesting workers until all tasks are flagged `Finished`. When the master marks all tasks as `Finished`, it exits the `while` loop immediately, save all the results, and calls `MPI_Abort` to kill all other processes and end the execution. `MPI_Abort` is called by the master to ensure that the execution terminates immediately as soon as all tasks are finished. It avoids hanging the execution indefinitely in a collective operation such as `MPI_Finalize` due to failed processes. Alternatively, implementations of fault-tolerant MPI, such as the User-Level Failure Mitigation [BBH+12] (ULFM), could be used to detect and exclude failed processes from the communicator. However, as such fault-tolerant implementations are not yet part of the MPI standard, the `MPI_Abort` approach is chosen instead, as it meets the requirement of immediate termination of execution, even in the case of failed processes.

The current implementation of rDLB works proactively and does not depend on, nor require fault detection. It does not add overhead to the application execution time in most cases, as the tasks are only rescheduled on free PEs when they wait for the completion of the execution after all tasks are scheduled. It only incurs overhead if the master is busy when all tasks are completed, such as FSC in Figure 13.6(a), which delays the termination of the application by the duration of the re-executed tasks on the master. The execution terminates as soon as all tasks are completed and report to the master successfully. A limitation of the current implementation is the master, being a single point of failure and contention. However, this limitation can be eliminated by transforming the *DLS4LB* into a decentralized library using approaches previously explored at the thread level [MEC18] or the process level [EC19b], which is expected to increase both its robustness and performance (see Section 9.2.2).

## 13.2 Experimental Evaluation and Discussion

A set of experiments are designed to test and evaluate the performance of scientific applications with rDLB under injected fatal and nonfatal perturbations in the computing system. The results are analyzed and discussed below.

### 13.2.1 Design of Experiments

A summary of the experiments performed to evaluate rDLB is presented in Table 13.1. In practice, PE or node failures are often noticed in long executions over a large number of PEs. For practical reasons, short applications execution times and small system size are used to show the benefit of the proposed approach.

**Applications** Two computationally-intensive applications are considered in this work: the parallel spin image algorithm (PSIA) [EFM+16] and the calculation of the Mandelbrot set [Man80]. Details and pseudocodes of the two applications are presented in Section 8.1. Both applications are parallelized at the process level only in these experiments, using MPI and the *DLS4LB*.

**Dynamic load balancing** The *DLS4LB* is employed for the self-scheduling of the loop iterations (tasks) in both applications. The *DLS4LB* is extended with the rDLB approach to enable robust scheduling as shown in Algorithm 13.1.

**Table 13.1 Design of factorial experiments to evaluate rDLB**

| Factors | Values | Properties |
|---|---|---|
| **Applications** | PSIA | $N = 20,000$ tasks[a]<br>Low variability among tasks |
| | Mandelbrot | $N = 262,144$ tasks<br>High variability among tasks |
| **Self-scheduling techniques** | STATIC | Static |
| | SS, FSC, mFSC, GSS,<br>TSS, FAC, WF | Dynamic nonadaptive<br>(with and without rDLB) |
| | AWF-B, -C, -D, -E, AF | Dynamic adaptive<br>(with and without rDLB) |
| **Execution scenarios** | Baseline | No failures or perturbations |
| **Failures** | one failure<br>$P/2$ failures[b]<br>$P-1$ failures | Assumptions:<br>(1) Fail-stop failures<br>(2) Failed cores do not recover<br>(3) Occur arbitrarily during execution |
| **Perturbations** | PE perturbations<br>Network latency perturbations<br>Combined PE and latency | A slow down of all PEs on a single node<br>Delay all communications of a single node<br>Combined |
| **Computing system** | miniHPC | 16 Dual socket Intel $E5-2640v4$ nodes<br>10 cores per socket<br>64 GB memory<br>nonblocking fat-tree topology |

[a] $N$ is the total number of tasks.
[b] $P$ is the total number of PEs.

**Injection of fatal and nonfatal perturbations** To simulate fail-stop failures, ranks make exit calls during the computation of the loop at arbitrary times during execution. To show the benefit of rDLB in tolerating a high number of failures, experiments are performed with 1, $P/2$, and $P-1$ failures, where $P$ is the number of PEs.

Nonfatal perturbations in PE availability are performed by running a CPU burner simultaneously on the same PEs as the running applications. For perturbations in network latency, MPI communication calls are intercepted through MPI profiling interface, and 10 seconds delays are added for any communication to or from a specified node. The injected delay is chosen to be long enough, such that it simulates a severe network latency perturbation. Given that the focus in this work is on computationally-intensive

applications, perturbations in network bandwidth have almost no effect on applications' performance (see Chapter 12).

**Computing system** The evaluation experiments are conducted on miniHPC (see Section 8.2.1). A total of 16 nodes are used in the experiments, with 16 ranks per node, eight ranks per socket, with a total of 256 ranks on 256 PEs.

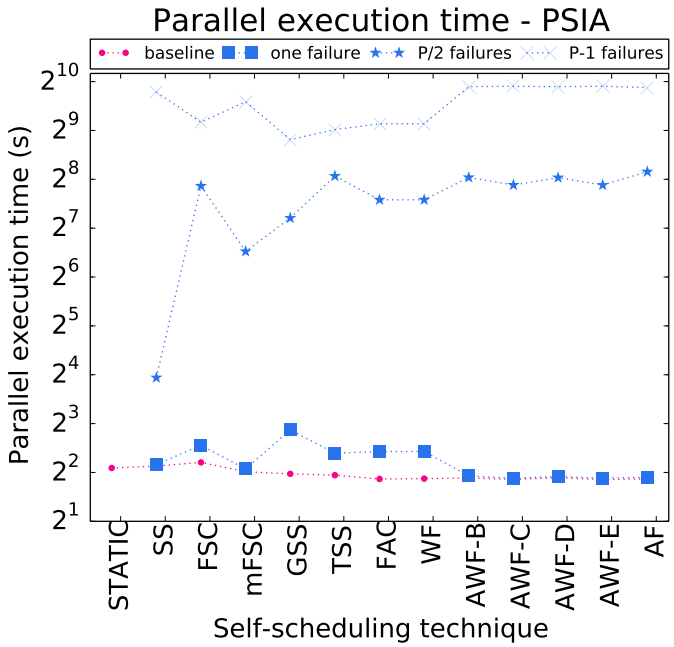
**Evaluation of robustness** The FePIA [AMS+04] procedure is applied to the performance results to evaluate the robustness of rDLB. The resilience is derived as $\rho_{res}(\phi, \pi)$, where $\phi$ is the parallel loop execution time $T_{par}$ and $\pi$ is the perturbation parameter considered, i.e., PE failures (see Section 3.2). $\rho_{res}(\phi, \pi)$ is calculated for applications execution with each DLS with $\pi_1$, $\pi_2$, and $\pi_3$, which corresponds to one PE failure, $P/2$ PE failures, and $P-1$ PE failures, respectively.

Similarly, flexibility, $\rho_{flex}(\phi, \pi)$ is calculated for applications' performances with each DLS under nonfatal perturbations, i.e., PE perturbations, latency perturbations, and combined perturbations for the three perturbations cases considered herein.

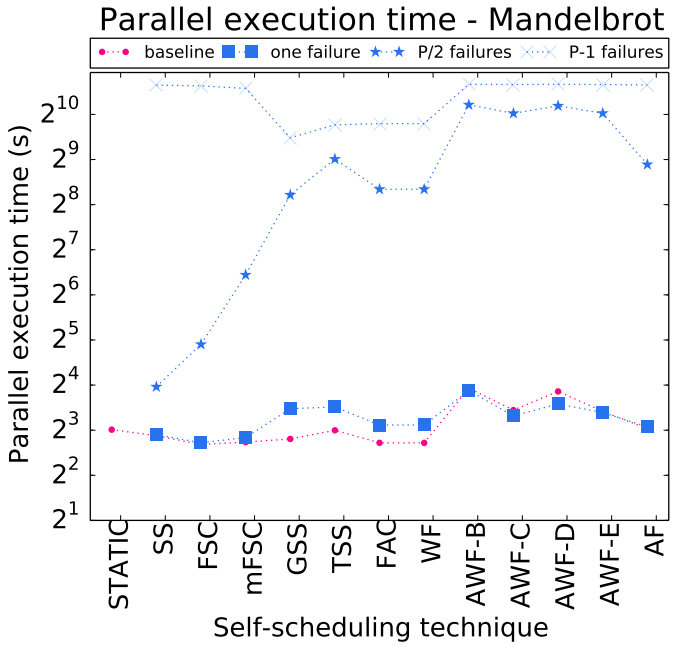
### 13.2.2 Evaluation and Discussion

The performance results of PSIA and Mandelbrot under failures and perturbations is shown in Figures 13.4 and 13.6 and the corresponding robustness analysis is presented in Figures 13.5 and 13.7. The STATIC technique is not included in the results with rDLB as rDLB applies only to dynamic self-scheduling techniques. The results show the average parallel loop execution time over 20 executions for each experiment. The coefficient of variation of parallel execution times in all experiments is below 0.64. As execution with *DLS4LB* without rDLB will not complete in the presence of failures, Figures 13.4(a) and 13.4(b) and Figure 13.5, show only the results with rDLB.

**Performance and resilience under failures** The inspection of Figures 13.4(a) and 13.4(b) and Figure 13.5 reveals that one PE failure is well tolerated with rDLB with almost no effect on the execution time, i.e., execution time with one failure is very close to the baseline. Specifically, for the adaptive DLS techniques, the execution times in the case of a single PE failure and baseline are qualitatively very close. The cost of tolerating $P/2$ failures depends significantly on the DLS technique. DLS techniques that assign small chunk sizes, such as SS (the most robust in this scenario), are more robust than techniques that assign large chunks. Small chunk sizes, in such cases, minimize the amount of lost work in the case of PE failures. In the case of $P-1$ failures, the work




(a) PSIA under PE failures




(b) Mandelbrot under PE failures

**Figure 13.4 Performance of PSIA and Mandelbrot with the rDLB under injected PE failures on miniHPC with 256 cores.** Execution with rDLB tolerates up to $P-1$ failures [MCC19].

is almost serialized on only the master, and in this case, the execution time depends on overhead of the DLS technique (number of scheduling rounds).

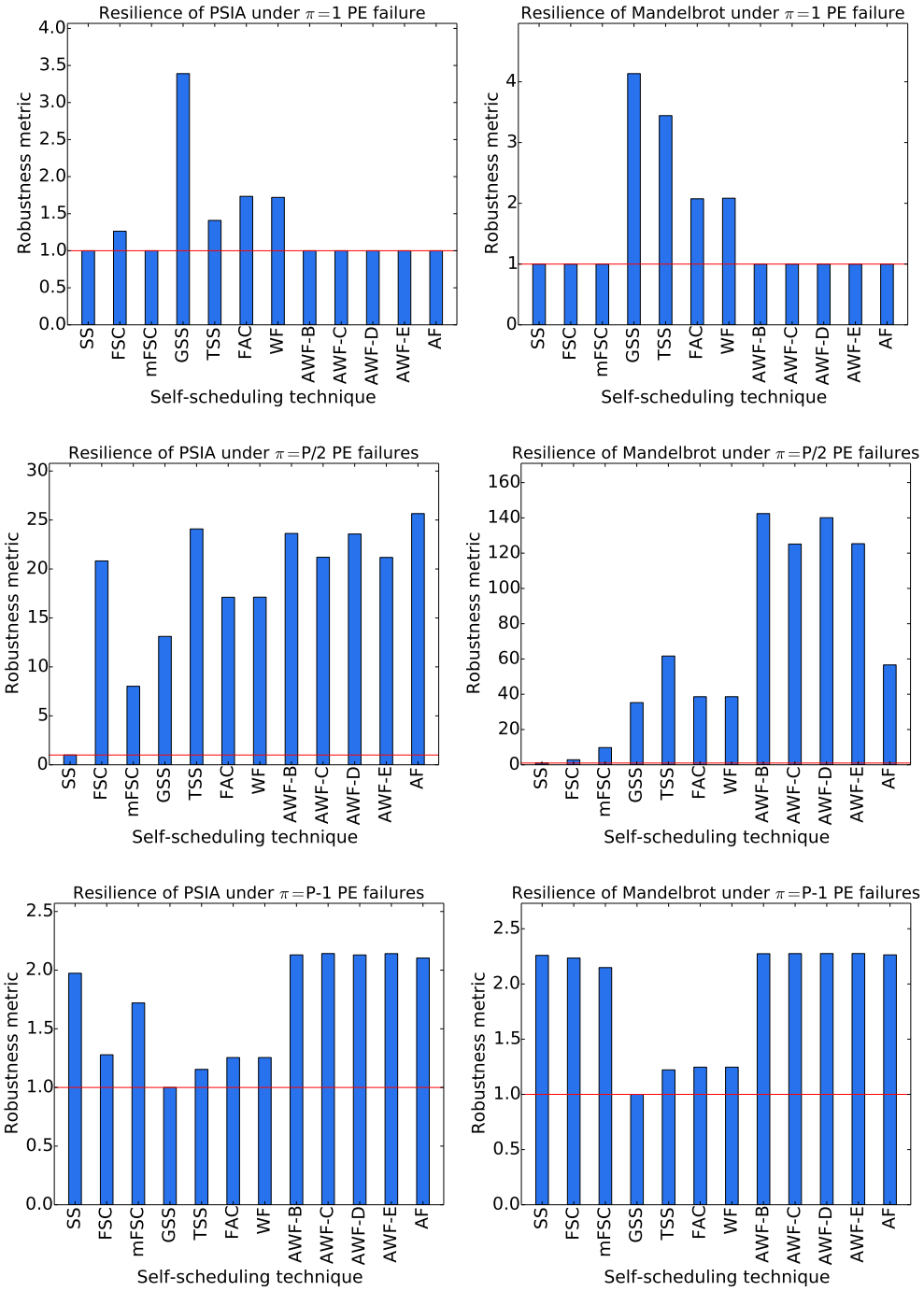
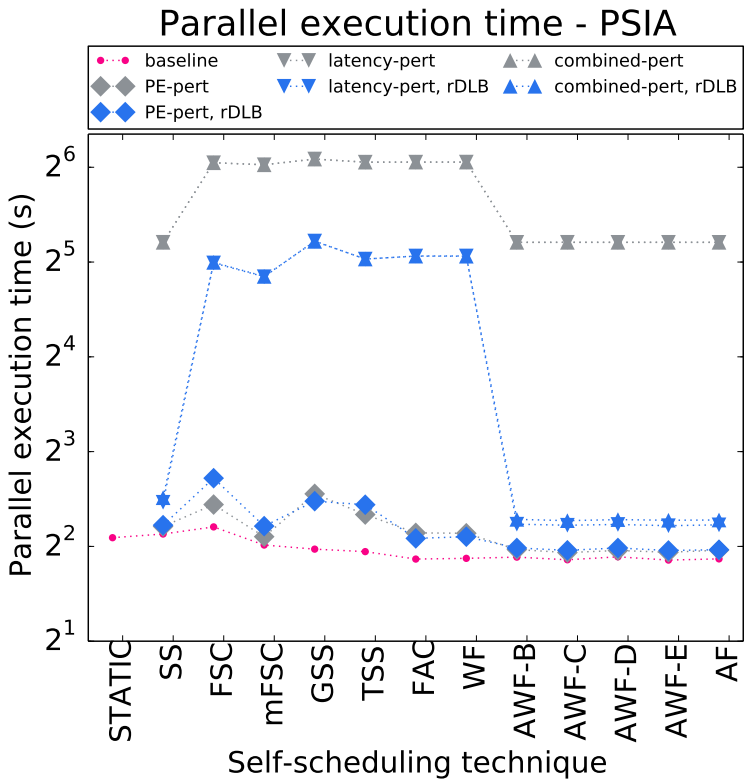


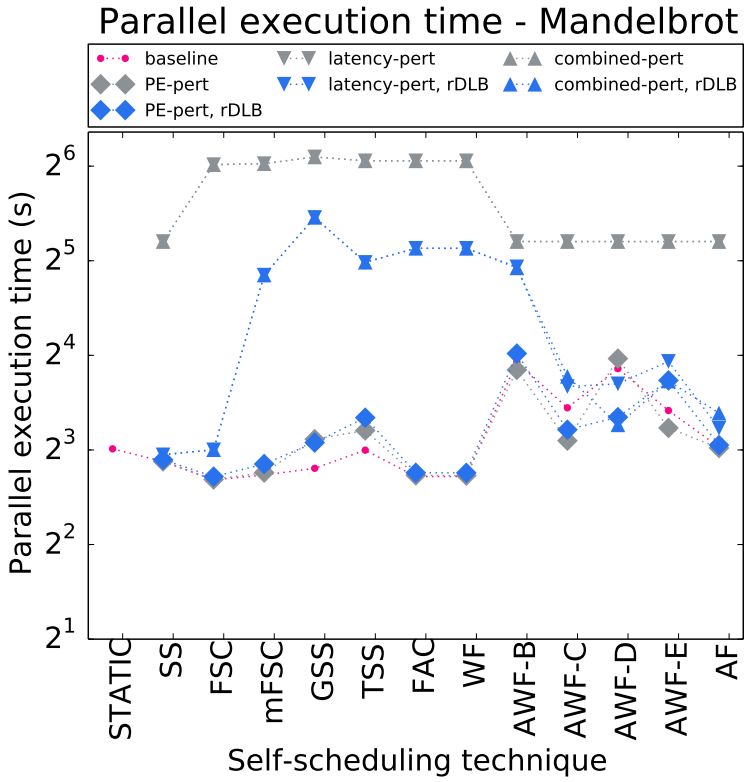

**Figure 13.5 Resilience of DLS techniques executing PSIA and Mandelbrot with the rDLB under failures on miniHPC with** 256 **cores.** The metrics show how many folds is a DLS technique robust with respect to the most robust DLS technique (red line, metric = 1) in a particular failure scenario (lower is better) [MCC19].

**Performance and flexibility under nonfatal perturbations** For the execution with perturbations, two experiments are performed per scheduling technique per perturbation scenario: without rDLB and with rDLB to show the benefit of rDLB in en-

hancing application performance and robustness under perturbations. The results show that perturbations in PE availability do not significantly affect performance. In mild perturbation scenarios such as PE perturbations, the execution with rDLB resulted in a slightly longer execution time than that without rDLB for the same DLS technique, e.g., FSC in Figure 13.6(a) vs. FSC in Figure 13.7 top-left, due to the re-execution of tasks on the master that delayed the termination of the application when all tasks are completed. Comparing the performance results when perturbations are injected in network latency and in combination with PE availability perturbations, one can see that rDLB enhanced the performance of PSIA in Figure 13.6(a) and Mandelbrot in Figure 13.6(b). Results of the flexibility metric in Figure 13.7 confirm that using the rDLB approach boosted the robustness of DLS techniques in case of severe latency perturbations and combined PE and latency availabilities perturbations. In fact, for the AWF-B, -C, -D, -E techniques, which are adaptive, their flexibility is enhanced by a factor greater than 30 by applying the rDLB approach in the case of combined PE and latency perturbations.




(a) PSIA under PE and network perturbations




(b) Mandelbrot under PE and network perturbations

**Figure 13.6 Performance of PSIA and Mandelbrot without (grey color) and with (blue color) the rDLB under injected PE perturbations and network latency perturbations on miniHPC with 256 cores.** Execution with rDLB enhanced the performance in the presence of perturbations by a factor of 7 with the adaptive DLS techniques under network latency perturbations [MCC19].

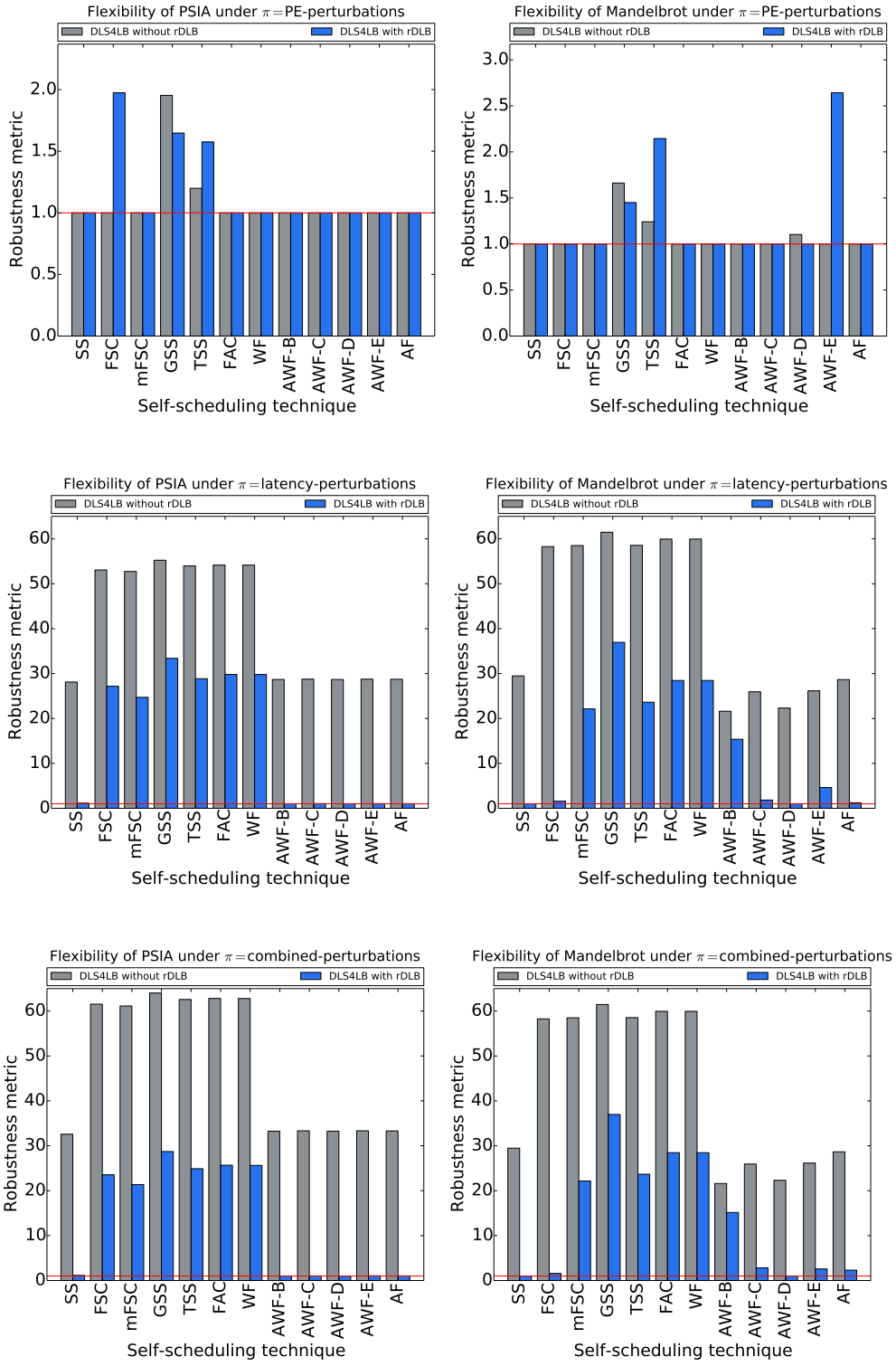



**Figure 13.7 Flexibility of DLS techniques executing PSIA and Mandelbrot without and with the rDLB under PE perturbations, and latency perturbation scenarios on miniHPC with 256 cores.** The metrics show how many folds is a DLS technique robust with respect to the most robust DLS technique (red line, metric = 1) in a particular perturbation scenario (lower is better) [MCC19].

Part V

# Conclusions and Outlook

# 14

# Conclusions and Outlook

In this chapter, we summarize and conclude the work presented in this thesis. We outline immediate and potential extensions of this work afterward.

## 14.1 Conclusions

In this doctoral thesis, we addressed the problem of robust dynamic load balancing of scientific applications on HPC systems. We proposed a realistic performance simulation approach for scientific applications with DLS. We discussed the influence of various parameters that affect the fidelity of the predicted simulative performance. *We proposed a realistic simulation approach for the accurate and fast simulation of MPI-based applications with minimal code changes.* Realistic performance simulation is an essential instrument in the analysis and optimization of the performance of scientific applications. *Realistic simulations result in similar analyses and conclusions to the analysis of the native results.* Based on simulation predictions of performance, confident decisions and selections can be made to improve the performance of applications on HPC systems.

*Realistic* performance simulations are *leveraged* for two goals in this work. First, realistic simulations were used for the reproduction of DLS experiments from literature to verify the present implementation of DLS techniques. To this end, we devised a method for the reproduction and verification of DLS techniques. The reproduction of such experiments increase the trustworthiness and eliminate uncertainties in the present implementation of DLS techniques. We were able to *reproduce the results in original publications within* 10% *average error.*

Second, realistic performance simulation is used to address the algorithm selection problem in the context of scheduling. Given the wide range of DLS techniques and their properties, the choice of the most efficient DLS technique is not trivial. Also, the study of the performance of scientific applications revealed that the most efficient DLS

technique for a certain application-system pair *changes* due to variations in system performance. Therefore, we devised and introduced SimAS, a simulation-assisted scheduling algorithm selection approach for the dynamic selection of DLS techniques under system perturbations. SimAS dynamically identifies and selects the most efficient DLS technique during execution based on the state of the system.

Following the verification of the present implementation of DLS techniques, we explored *centralized and decentralized* coordination approaches in implementing DLS techniques. We discussed the advantages and limitations of each approach. We presented *DLS implementations* in the most successful and widely used programming models in HPC, namely *OpenMP* and *MPI*. This *encourages scientific applications developers* to employ DLS techniques, which is crucial for improving the performance and scalability of applications on large-scale HPC systems.

We showed how to use DLS techniques for the dynamic load balancing at a *single or two levels* of software parallelism. Given the increased complexity of the present and future HPC systems, most scientific applications employ multilevel of software parallelism to harness the computational power of modern HPC systems. We focus on the process and thread levels of parallelism, specifically MPI+OpenMP applications, as they represent the most commonly used parallelization method in scientific applications. Experimental evaluation of two-level load balancing using DLS shows that only load balancing at the two levels can achieve the best performance. Two-level dynamic load balancing improved *production astrophysics application performance by* 15% *and other applications by up to* 21%. Interestingly, we found that *load balancing at one level can affect the load imbalance at the other level and vice versa.* Also, we found that *the best performing two-level combination is not always the combination of the two best performing DLS techniques at a single level alone.*

To address *nonfatal and fatal* perturbations in large scale HPC systems, we devised rDLB for the robust dynamic load balancing of applications. rDLB *proactively* reschedules tasks to tolerate fatal and nonfatal perturbations. It does *not require any fault detection* mechanism. The analytical analysis of rDLB shows that fault tolerance overhead decreases with increasing the number of PEs, which is a desirable property for the ever-increasing scale of HPC systems. We integrated rDLB into the *DLS4LB* MPI-based DLS library. Experimental evaluation of rDLB shows that it *tolerates* up to $P-1$ *failures*, where $P$ is the number of PE allocated to the application and *boosts the robustness* of DLS techniques by up to a factor of 30 under nonfatal perturbations.

In summary, we provide methods for the fast and accurate performance simulation of applications and reproduction for verification of DLS experiments. We study the performance of applications under perturbations and show that *robustness does not imply*

*efficiency*. Not only perturbations in the computing power are considered, but for the first time, also perturbations in the network (bandwidth and latency) are examined for their effects on performance. We proposed SimAS for the dynamic selection of DLS techniques during execution to solve the algorithm selection problem in the context of scheduling under perturbations. We devised rDLB to address fatal and nonfatal perturbations during execution. For the first time, we enriched applications parallelized via MPI with robust DLS techniques.

Modern and future HPC systems are characterized by their large scale, heterogeneity, and high performance variability. Also, large components count in systems leads to high failure rates that may render these systems unusable. Therefore, robust scheduling methods are of paramount importance to achieve a load balanced, improved performance in the presence of system variability and failures. We proposed robust dynamic load balancing methods and integrated them into OpenMP and MPI to fill the gap and improve the performance of applications. We studied the interplay between load balancing at the thread and process level with DLS techniques. Verified DLS techniques, realistic simulations, SimAS and rDLB that are integrated into an MPI-based dynamic load balancing library (*DLS4LB*), provide the needed tools for robust load balanced performance of scientific applications on HPC systems.

## 14.2 Outlook

This doctoral thesis opens possible extensions in various research directions. In the reproduction and verification direction, our approach can be employed for the reproduction of various experiments for the verification of additional DLS techniques.

In the DLS techniques implementation direction, the MPI-based DLS library *DLS4LB* is implemented in a centralized coordination approach in this work. A decentralized coordination approach is planned to be applied in the immediate future for the implementation of such techniques at the MPI, similar to OpenMP implementation. Also, the extension of commonly used programming models and runtime libraries with DLS techniques other than MPI and OpenMP, such as Charm++ using its load balancing interface [KK93] is planned in the future. Since loop iterations are in fact independent tasks, it is vital to employ DLS techniques to extend tasking programming approaches, such as Cilk [BJK+96], TBB [Phe08], OmpSs [BMD+11], HPX [KHAL+14], and Habanero [BBC+09], which only support work-stealing and no self-scheduling techniques, to extend the benefit of DLS to their users as well.

In the scheduling algorithm selection direction, the study of perturbations and their influence on the performance of applications in native and simulative experiments can

be used to synthesize a set of selection rules that replaces the need for online simulation in SimAS. Using this set of rules will enable applications to select the most efficient DLS technique dynamically during execution.

In the robust scheduling direction, rDLB can be integrated into more scheduling libraries that employ self-scheduling, such as OpenMP. Herein, rDLB is integrated into *DLS4LB* at the MPI level.

In the multi-level scheduling direction, a more in-depth study and analysis of the interplay of load balancing between various levels of software parallelism, including batch level and application level (process and thread) is on the top of priorities as an immediate next step for this work. The cooperation between schedulers at various levels can achieve higher degrees of load balance and robust performance than that achievable by a single level only.

Exascale systems are approaching, and the ever-increasing scale of HPC systems brings challenges that require holistic solutions. The scalability of decentralized DLS implementations, realistic performance simulations, dynamic scheduling technique selection, and robust multi-level scheduling are all seen as necessary research and production instruments for Exascale and beyond HPC systems. This thesis opens up the horizons into understanding the interplay of load balancing between various levels of software parallelism and lays the ground for robust multi-level scheduling to address unprecedented load imbalance and perturbations foreseen in Exascale HPC systems.

# A

# Notation and Terminology

Here we summarize the symbols and terms used in this thesis and their definition.

**Table A.1 Symbols**

| Symbol | Description |
|---|---|
| $N$ | Total number of tasks or loop iterations |
| $P$ | Number of processing elements (PEs) |
| $R$ | Remaining number of unscheduled tasks |
| $\mu$ | Mean of task execution times |
| $\sigma$ | Standard deviation of task execution times |
| $h$ | Scheduling overhead |
| $i$ | PE ID |
| $j$ | Scheduling step |
| $w_i$ | Relative weight of PE $i$ |
| $n_{ij}$ | Chunk size allocated to PE $i$ at scheduling step $j$ |
| $t_{ij}$ | Execution time of $n_{ij}$ on PE $i$ |
| $f$ | First chunk size |
| $l$ | Last chunk size |
| $D$ | Decrement between consecutive chunks |
| $\theta_i$ | Weighted average ratio |
| $\bar{\theta}_i$ | Average weighted average ratio |
| $\eta_i$ | PE raw computational weight |
| $\hat{\eta}$ | Sum of PE raw computational weights |
| $\phi$ | Performance feature of interest |
| $\pi$ | Perturbation parameter |
| $\rho_{res}$ | Resilience metric of a DLS technique |
| $\rho_{flex}$ | Flexibility metric of a DLS technique |
| $r_{DLS}$ | Robustness radius, indicating how much a performance feature $\phi$ was affected by a perturbation $\pi$ |
| $r_{minDLS}$ | Minimum robustness radius |

**Table A.2 Terms used in this thesis**

| Term | Description |
|---|---|
| Batch of chunks | Chunks of tasks that are calculated at the same scheduling step |
| Centralized coordination | A management of parallelism scheme, where a thread or process is responsible for the coordination between all other threads or processes in the application |
| Chunk size | The number of tasks assigned to a PE at a scheduling step |
| Computationally-intensive applications | Applications that spend most of their execution time computing, improving speed of computations in these applications improves their overall performance (compute-bound) |
| Decentralized coordination | A management of parallelism scheme, where entities of an application coordinate with each other by sharing certain control data without the need of dedicating one entity to perform such task |
| DLS | Dynamic loop self-scheduling |
| Error | An incorrect value or state, it occurs when a faulty unit or value is used |
| Fatal perturbations or failures | A failure represents a stop of service of certain system component or application. A fatal perturbation causes the applications running on the system to fail |
| Fault | A sudden malfunction that occurs in a computing system, such as a bit flip or incorrect control signal |
| Fault tolerance | A property of an application or a system, where faults are transparent to the users. The application or system can continue correct operation seamlessly in the presence of faults |
| Flexibility | A measure of the robustness against nonfatal perturbations |
| FLOP | A floating-point operation, e.g., a single addition or multiplication on two floating-point numbers |
| FLOP count | The amount of floating-point operations in a task |
| FLOP/s | The number of attained floating-point operations per second as a measure of performance |
| Hard error | A persistent errors, typically caused by hardware damage. To recover from hard errors, the faulty component needs to be replaced or fixed. |
| Heterogeneous system | Computing systems with components, which perform the exact function, but with different properties, e.g. different types of CPUs, CPU+GPU |
| HPC | High performance computing |
| Homogeneous system | Computing systems with components of the same kind and same properties |
| Load imbalance | Uneven computational load on parallel executing PEs that leads to uneven finishing times of PEs and degraded performance |
| MTBF | Mean time between failures, MTBF = MTTF+MTTR |
| MTTF | Mean time to failure, it measures the expected operational time of a system or component before a halt due to failure |
| MTTR | Mean time to repair, it measures how much time is needed to recover from a failure |
| Multi-level scheduling | Scheduling on more than one software parallelism level, e.g. thread, process, batch. Schedulers on all software level cooperate to achieve load balance and improve performance |

**Table A.3 Terms used in this thesis (continued)**

| Term | Description |
|---|---|
| Native experiments | Direct performance experiments obtained by executing applications on HPC systems |
| Nonfatal perturbations | Errors or system noise that interfere with and impact the performance of a running application without causing it to fail |
| Parallel loop execution time | The finishing time of the latest PE |
| PE | A processing element, e.g. a CPU core |
| Perturbations | Interference that impacts applications performance without causing the failure of the application (nonfatal perturbations) or causes the failure of an application (fatal perturbations) |
| Realistic simulations | Simulations that captures performance characteristics of native experimentation and results in a performance analysis similar to that based on the native results |
| Resilience | A measure of the robustness against failures |
| Robustness | Maintaining a certain performance level in the presence of perturbations |
| Scheduling overhead | Extra-time spent in calculating and assigning chunks of tasks to PEs |
| Scheduling step or scheduling round | Calculation and assignment of a chunk of tasks to PEs |
| SDC | Silent data corruption, an error that occurs in applications data and/or code and is undetected. It can lead to false results |
| Simulative experiments | Opposite of native, experimentation or results obtained from simulation |
| Single-level scheduling | Scheduling at only one level of software parallelism |
| Single-sweep applications | Opposite of time-stepping applications. Applications that compute certain operations without updating the application state in between |
| Soft errors | Transient errors, therefore, they can not be reproduced |
| Strong scaling | Increasing the number of PEs while fixing the problem size. It tests how much speedup can be obtained by allocating more resources to a certain application |
| System variability | System noise, background processes, and operating system noise that perturb a running application and causes its variable performance across different executions |
| Time-stepping applications | Applications that repeatedly calculate certain operations in a loop. At each iteration of the loop (time-step), the application state is updated based on the calculations in the previous iteration(s) |
| TiT | Time-independent trace, an execution trace that contains computations represented as FLOP count per task and communications represented as bytes instead of timings |
| Two-level scheduling | Scheduling at two levels of software parallelism, e.g. process and thread levels. Both levels cooperate to achieve load balanced improved performance |
| Weak scaling | Increasing problem size and the number of PEs allocated to an application by keeping their ratio fixed. It tests how large can a problem be solved by allocating more resources to the application |

# Bibliography


[ACM16] ACM. Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-badging. [Online; accessed 24 October 2017]. 2016.

[AMS+04] Shoukat Ali, Anthony A Maciejewski, Howard Jay Siegel, and Jong-Kook Kim. Measuring the Robustness of a Resource Allocation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):630–641, 2004.

[ABC+88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Parallel and Distributed Computing*, 5(5):617–640, 1988.

[Lap] An Enhanced OpenMP Library. https://github.com/lapesd/libgomp. Accessed 2019-07-30. 2019.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View From Berkeley. Technical report. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, 2006.

[AE18] Rizwan A Ashraf and Christian Engelmann. Analyzing the Impact of System Reliability Events on Applications in the Titan Supercomputer. In *Proceedings of the IEEE/ACM 8th Workshop on Fault Tolerance for HPC at Extreme Scale*, 2018, pages 39–48.

[ADE+01] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*. Springer, 2001, pages 1–10.

[ADADB+ne] Jürgen Assfalg, Gianpaolo D Amico, Alberto Del Bimbo, and Pietro Pala. 3D Content-Based Retrieval with Spin-images. In *Proceedings of the International Conference on Multimedia and Expo*, June 2004, pages 771–774.

[ATN+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[Bai11] David H Bailey. NAS Parallel Benchmarks. *Encyclopedia of Parallel Computing*:1254–1259, 2011.

[BGB+19] Seonmyeong Bak, Yanfei Guo, Pavan Balaji, and Vivek Sarkar. Optimized Execution of Parallel Loops via User-Defined Scheduling Policies. In *Proceedings of the 48th International Conference on Parallel Processing*. ACM, 2019.

[BMW+18] Seonmyeong Bak, Harshitha Menon, Sam White, Matthias Diener, and Laxmikant Kale. Multi-Level Load Balancing with an Integrated Runtime Approach. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2018, pages 31–40.

[BBC13] Mahadevan Balasubramanian, Ioana Banicescu, and Florina M. Ciorba. Analyzing the Robustness of Scheduling Algorithms using Divisible Load Theory on Heterogeneous Systems. In *Proceedings of the 12th International Symposium on Parallel and Distributed Computing*. IEEE, 2013, pages 45–52. ISBN: ISBN-13 9780769550183.

[BBC14] Mahadevan Balasubramanian, Ioana Banicescu, and Florina M. Ciorba. Robustness Prediction and Evaluation of Divisible Load Scheduling on Heterogeneous Systems with Uncertain Perturbations. In *Proceedings of the 13th International Symposium on Parallel and Distributed Computing*. IEEE, 2014.

[BC05] Ioana Banicescu and Ricolindo L. Cariño. Addressing the Stochastic Nature of Scientific Computations via Dynamic Loop Scheduling. *Electronic Transactions on Numerical Analysis*, 21:66–80, 2005.

[BCCn09] Ioana Banicescu, Florina M. Ciorba, and Ricolindo L. Cariño. Towards the Robustness of Dynamic Loop Scheduling on Large-Scale Heterogeneous Distributed Systems. In *Proceedings of the 8th International Symposium on Parallel and Distributed Computing*, July 2009, pages 129–132.

[BCS13] Ioana Banicescu, Florina M. Ciorba, and Srishti Srivastava. *Scalable Computing: Theory and Practice*. In. (Chapter 22). John Wiley & Sons, Inc, 2013. part Performance Optimization of Scientific Applications using an Autonomic Computing Approach, pages 437–466.

[BFH95] Ioana Banicescu and Susan Flynn Hummel. Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 1995, pages 43–43.

[BL00] Ioana Banicescu and Zhijun Liu. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In *Proceedings of the High Performance Computing Symposium*, 2000, pages 122–129.

[BV02] Ioana Banicescu and Vijay Velusamy. Load Balancing Highly Irregular Computations with the Adaptive Factoring. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium Workshops*, 2002, page 195.

[BVD03] Ioana Banicescu, Vijay Velusamy, and Johnny Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications With Adaptive Weighted Factoring. *Cluster Computing*, 6(3):215–226, 2003.

[BBC+09] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 2009, pages 735–736.

[BGTK+11] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pages 1–12.

[BEDG19] Olivier Beaumont, Lionel Eyraud-Dubois, and Yihong Gao. Influence of Tasks Duration Variability on Task-Based Runtime Schedulers. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2019, pages 16–25.

[BCR+16] Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun. Assessing General Purpose Algorithms to Cope with Fail-stop and Silent Errors. *ACM Transactions on Parallel Computing*, 3(2):13, 2016.

[BB87] Marsha J. Berger and Shahid H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, (5):570–580, 1987.

[BBC+] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15 (2008).

[BBGD+17] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Toward General Software Level Silent Data Corruption Detection for Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3642–3655, 2017.

[BGG18] Luke Bertot, Stéphane Genaud, and Julien Gossa. Improving Cloud Simulation Using the Monte-Carlo Method. In *Proceedings of the European Conference on Parallel Processing*. Springer, 2018, pages 404–416.

[BBH+12] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An Evaluation of User-level Failure Mitigation Support in MPI. In *European MPI Users' Group Meeting*. Springer, 2012, pages 193–203.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[BL99] Robert D Blumofe and Charles E Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[BGA+16] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the Root Causes of Wait States in Large-scale Parallel Applications. *ACM Transactions on Parallel Computing*, 3(2):11, 2016.

[BGW+10] David Bohme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the Root Causes of Wait States in Large-scale Parallel Applications. In *Proceedings of the 39th International Conference on Parallel Processing*. IEEE, 2010, pages 90–100.

[BWG12] David Böhme, Felix Wolf, and Markus Geimer. Characterizing Load and Communication Imbalance in Large-scale Parallel Applications. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pages 2538–2541.

[BBB+14] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Frank Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified Model for Assessing Checkpointing Protocols at Extreme Scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[BDD+09] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based Fault Tolerance Applied to High Performance Computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[BBC+17] Anthony Boulmier, Ioana Banicescu, Florina M. Ciorba, and Nabil Abdennadher. An Autonomic Approach for the Selection of Robust Dynamic Loop Scheduling Techniques. In *Proceedings of 16th International Symposium on Parallel and Distributed Computing*, 2017, pages 9–17.

[BWA16] Anthony Boulmier, John White, and Nabil Abdennadher. Towards a Cloud Based Decision Support System for Solar Map Generation. In *IEEE International Conference on Cloud Computing Technology and Science*, 2016, pages 230–236.

[BM09] Greg Bronevetsky and Adam Moody. Scalable I/O Systems via Node-local Storage: Approaching 1 TB/sec File I/O. Technical report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2009.

[BDG+00] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[BCR91] Raymond M. Bryant, Herbert Y. Chang, and Bryan S. Rosenburg. Operating System Support for Parallel Programming on RP3. *IBM Journal of Research and Development*, 35(5.6):617–634, 1991.

[Bud17] Patrick Buder. Evaluation and Analysis of Dynamic Loop Scheduling in OpenMP. https://hpc.dmi.unibas.ch/HPC/Completed_ Theses_ and_ Projects_ files/2018_ Patrick_ Buder_ MA_ ThesisJanuary2018.pdf. [Online; accessed 24 July 2019]. 2017.

[BMD+11] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. Productive Cluster Programming with OmpSs. In *Proceedings of the European Conference on Parallel Processing*. Springer, 2011, pages 555–566.

[CGSF17] Rubén M. Cabezón, Domingo Garcia-Senz, and Joana Figueira. SPHYNX: An Accurate Density-based SPH Method for Astrophysical Applications. *Astronomy & Astrophysics*, 606:A78, 2017.

[CGR08] Rubén M. Cabezón, Domingo García-Senz, and Antonio Relaño. A One-parameter Family of Interpolating Kernels for Smoothed Particle Hydrodynamics Studies. *Journal of Computational Physics*, 227:8523–8540, October 2008.

[CJ10] Louis-Claude Canon and Emmanuel Jeannot. Evaluation and Optimization of the Robustness of DAG Schedules in Heterogeneous Environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):532–546, 2010.

[CB07] Ricolindo L. Cariño and Ioana Banicescu. A Tool for a Two-level Dynamic Load Balancing Strategy in Scientific Applications. *Scalable Computing: Practice and Experience*, 8(3), 2007.

[CB08] Ricolindo L. Cariño and Ioana Banicescu. Dynamic Load Balancing With Adaptive Factoring Methods in Scientific Applications. *Journal of Supercomputing*, 44(1):41–63, 2008.

[CGL+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[CBL08] Marc Casas, Rosa Badia, and Jesús Labarta. Automatic Analysis of Speedup of MPI Applications. In *Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, 2008, pages 349–358.

[CC19] Aurélien Cavelan and Florina M. Ciorba. Algorithm-Based Fault Tolerance for Parallel Stencil Computations. In *Proceedings of the International Conference on Cluster Computing*. IEEE, Albuquerque, September 2019.

[CL85] K Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009, pages 44–54.

[Che13] Zizhong Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *ACM SIGPLAN Notices*. Volume 48. (8), 2013, pages 167–176.

[CK13] K.-S. Choi and D.-H. Kim. Angular-partitioned Spin-image Descriptor for Robust 3D Facial Landmark Detection. *Electronics Letters*, 49(23):1454–1455, 2013.

[Cio17] Florina M. Ciorba. The Importance and Need for System Monitoring and Analysis in HPC Operations and Research. In *Proceedings of the 3rd bwHPC-Symposium*. Heidelberg, 2017, 10 pp.

[CIB18] Florina M. Ciorba, Christian Iwainsky, and Patrick Buder. OpenMP Loop Scheduling Revisited: Making a Case for More Schedules. In *Proceedings of the 2018 International Workshop on OpenMP*, September 2018.

[CRA+08] Florina M. Ciorba, Ioannis Riakiotakis, Theodore Andronikos, George Papakonstantinou, and Anthony T. Chronopoulos. Enhancing Self-scheduling Algorithms via Synchronization and Weighting. *Journal of Parallel and Distributed Computing*, 68(2):246–264, 2008. ISSN: 0743-7315.

[CLH19] Tom Cornebize, Arnaud Legrand, and Franz Heinrich. Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study. Working Paper or Preprint. 2019.

[DSCB03] Daniel Paranhos Da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-tasks Applications on Computational Grids. In *Proceedings of the European Conference on Parallel Processing*. Springer, 2003, pages 169–180.

[Dal06] John T. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[DMS12] Frederic Desprez, George S Markomanolis, and Frédéric Suter. Improving the Accuracy and Efficiency of Time-independent Trace Replay. In *Proceedings of the ACM/IEEE International High Performance Computing, Networking, Storage and Analysis*, November 2012, pages 446–455.

[DBK06] Karen D. Devine, Erik G. Boman, and George Karypis. Partitioning and Load Balancing for Emerging Parallel Applications and Architectures. In, *Parallel Processing for Scientific Computing*, pages 99–126. SIAM, 2006.

[DGG+19] Sheng Di, Hanqi Guo, Rinku Gupta, Eric R. Pershey, Marc Snir, and Franck Cappello. Exploring Properties and Correlations of Fatal Events in a Large-Scale HPC System. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):361–374, February 2019. ISSN: 1045-9219.

[DGS+17] Sheng Di, Rinku Gupta, Marc Snir, Eric Pershey, and Franck Cappello. Logaider: A Tool for Mining Potential Correlations of HPC Log Events. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2017, pages 442–451.

[DGK19] Jack Dongarra, Steven Gottlieb, and William T. C. Kramer. Race to Exascale. *Computing in Science Engineering*, 21(1):4–5, 2019. ISSN: 1521-9615.

[DHR15] Jack Dongarra, Thomas Herault, and Yves Robert. Fault Tolerance Techniques for High Performance Computing. In, *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[DLP03] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[DRS05] Antonio J Dorta, Casiano Rodriguez, and Francisco de Sande. The OpenMP Source Code Repository. In *Proceedings of the IEEE 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2005, pages 244–250.

[DBB+12] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. *ACM SigPlan Notices*, 47(8):225–234, 2012.

[ESS13] Nosayba El-Sayed and Bianca Schroeder. Reading Between the Lines of Failure Logs: Understanding How HPC Systems Fail. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013, pages 1–12.

[EC19a] Ahmed Eleliemy and Florina M. Ciorba. Dynamic Loop Scheduling Using MPI Passive-Target Remote Memory Access. In *Proceedings of the 27th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2019.

[EC19b] Ahmed Eleliemy and Florina M. Ciorba. Hierarchical Dynamic Loop Self-Scheduling on Distributed-Memory Systems Using an MPI+MPI Approach. In *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum, 20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2019.

[EFM+16] Ahmed Eleliemy, Mahmoud Fayze, Rashid Mehmood, Iyad Katib, and Naif Aljohani. Load-balancing on Parallel Heterogeneous Architectures: Spin-image Algorithm on CPU and MIC. In *Proceedings of the 9th EUROSIM Congress on Modelling and Simulation*, 2016, pages 623–628.

[EMC16] Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Simulating Batch and Application Level Scheduling Using GridSim and SimGrid. Poster at ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis. November 2016.

[EMC17a] Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Efficient Generation of Parallel Spin-images Using Dynamic Loop Scheduling. In *Proceedings of the 19th IEEE International Conference for High Performance Computing and Communications Workshops*, 2017, pages 34–41.

[EMC17b] Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Exploring the Relation Between Two Levels of Scheduling Using a Novel Simulation Approach. In *Proceedings of 16th International Symposium on Parallel and Distributed Computing (ISDPC)*, July 2017, page 8.

[EAW+02] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A Survey of Rollback-recovery Protocols in Message Passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[EOS09] Christian Engelmann, Hong Ong, and Stephen L. Scott. The Case for Modular Redundancy in Large-scale High Performance Computing Systems. In *Proceedings of the IASTED International Conference*. Volume 641, 2009, page 046.

[EWG+11] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *Proceedings of the International Conference on Parallel Computing*, 2011, pages 481–490.

[FD00] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2000, pages 346–353.

[FRO+11] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and Ron Brightwell. rMPI: Increasing Fault Resiliency in a Message-passing Environment. *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011.

[FSLI+11] Kurt Ferreira, Jon Stearley, James H Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, page 44.

[FME+12a] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large Scale High Performance Computing. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, page 78.

[FME+12b] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, Utah, 2012, 78:1–78:12. ISBN: 978-1-4673-0804-5.

[FHSU+96] Susan Flynn Hummel, Jeanette Schmidt, RN Uma, and Joel Wein. Load-sharing in Heterogeneous Systems via Weighted Factoring. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996, pages 318–328.

[FHSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, 1992.

[FX07] Song Fu and Cheng-Zhong Xu. Exploring Event Correlation for Failure Prediction in Coalitions of Clusters. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, 2007, page 41.

[GCS+12] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault Prediction Under the Microscope: A Closer Look into HPC Systems. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pages 1–11.

[GGA+17] Luis A. García-González, César R. García-Jacas, Liesner Acevedo-Martínez, Rafael A. Trujillo-Rasúa, and Dirk Roose. Self-Scheduling for a Heterogeneous Distributed Platform. In *Proceedings of the International Conference on Parallel Computing*, 2017, pages 232–241.

[GCE12] Domingo García-Senz, Rubén M. Cabezón, and Jose Antonio Escartín. Improving Smoothed Particle Hydrodynamics with an Integral Approach to Calculating Gradients. *Astronomy & Astrophysics*, 538:A9, A9, February 2012.

[Gei11] A Geist. What is the Monster in the Closet? In *Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking*. Volume 2. (4), 2011.

[GSG+16] Mark Gottscho, Mohammed Shoaib, Sriram Govindan, Bikash Sharma, Di Wang, and Puneet Gupta. Measuring the Impact of Memory Errors on Application Performance. *IEEE Computer Architecture Letters*, 16(1):51–55, 2016.

[GCC+19] Danilo. Guerrera, Aurélien. Cavelan, Rubén. M. Cabezón, David. Imbert, Jean G. Piccinali, Ali Mohammed, Lucio Mayer, Darren S. Reed, and Florina M. Ciorba. SPH-EXA: Enhancing the Scalability of SPH codes Via an Exascale-Ready SPH Mini-App. In *Proceedings of the 2019 Spheric International Workshop*. Exeter, June 2019.

[GPE+17] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in Large Scale Systems: Long-term Measurement, Analysis, and Implications. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, page 44.

[Gut18] Samuel K Gutiérrez. Adaptive Parallelism for Coupled, Multithreaded Message-Passing Programs. PhD thesis. University of New Mexico, 2018.

[Hag97] Torben Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journal of Parallel and Distributed Computing*, 47(2):185–197, 1997.

[HD06] Paul H. Hargrove and Jason C Duell. Berkeley Lab Checkpoint/restart (blcr) for Linux Clusters. In *Journal of Physics: Conference Series*. Volume 46. (1). IOP Publishing, 2006, page 494.

[HKR+12] Daniel F Harlacher, Harald Klimach, Sabine Roller, Christian Siebert, and Felix Wolf. Dynamic Load Balancing for Unstructured Meshes on Space-filling Curves. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pages 1661–1669.

[HLK03] Chao Huang, Orion Lawlor, and Laxmikant V. Kale. Adaptive MPI. In *International workshop on languages and compilers for parallel computing*. Springer, 2003, pages 306–322.

[HA84] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, 100(6):518–528, 1984.

[HT13] Sascha Hunold and Jesper Larsson Träff. On the State and Importance of Reproducible Experimental Research in Parallel Computing. *Computing Research Repository*, abs/1308.3648, 2013.

[HSM+07] Joshua Hursey, Jeffrey M. Squyres, Timothy I Mattox, and Andrew Lumsdaine. The Design and Implementation of Checkpoint/restart Process Fault Tolerance for OpenMPI. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, 2007, pages 1–8.

[HLK+97] Ali R Hurson, Joford T Lim, Krishna M Kavi, and Ben Lee. Parallelization of Doall and Doacross Loops - A Survey. In, *Advances in Computers*. Volume 45, pages 53–103. Elsevier, 1997.

[JFDM+17] Saurabh Jha, Valerio Formicola, Catello Di Martino, Mark Dalton, William T. Kramer, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Resiliency of HPC Interconnects: A Case Study of Interconnect Failures and Recovery in Blue Waters. *IEEE Transactions on Dependable and Secure Computing*, 15(6):915–930, 2017.

[JJM+11] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High Performance Computing Using MPI and OpenMP on Multicore Parallel Systems. *Parallel Computing*, 37(9):562–575, 2011.

[JH99] Andrew E. Johnson and Martial Hebert. Using Spin-images for Efficient Object Recognition in Cluttered 3D Scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(5):433–449, 1999.

[Joh97] Andrew Edie Johnson. Spin-Images: A Representation for 3D Surface Matching. PhD thesis. Robotics Institute, Carnegie Mellon University, 1997.

[KHAL+14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, page 6.

[Kal19] Gustav Kalbe. The European Approach to the Exascale Challenge. *Computing in Science & Engineering*, 21(1):42–47, 2019.

[KAB+12] Laxmikant V Kale, Anshu Arya, Abhinav Bhatele, Abhishek Gupta, Nikhil Jain, Pritish Jetley, Jonathan Lifflander, Phil Miller, Yanhua Sun, Ramprasad Venkataramanz, Lukasz Wesolowski, and Gengbin Zheng. Charm++ for Productivity and Performance. Technical report. Technical Report at the 2011 HPC class II challenge, 2012.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. Association for Computing Machinery, 1993, pages 91–108.

[KIK+19] Vivek Kale, Christian Iwainsky, Michael Klemm, Jonas Kondorfer, and Florina M. Ciorba. Toward a Standard Interface for User-Defined Scheduling in OpenMP. *Arxiv preprint arxiv:1906.08911*, 2019.

[KRG14] Vivek Kale, Amanda Randles, and William D. Gropp. Locality-optimized Mixed Static/Dynamic Scheduling for Improving Load Balancing on SMPs. In *Proceedings of the 21st European MPI Users' Group Meeting*. ACM, 2014, page 115.

[KTV+19] Franziska Kasielke, Ronny Tscüter, Markus Velten, Florina M. Ciorba, Christian Iwainsky, and Ioana Banicescu. Exploring Loop Scheduling Enhancements in OpenMP: An LLVM Case Study. In *Proceedings of the 18th International Symposium on Parallel and Distributed Computing*, 2019.

[KTMSL+17] Rafael Keller Tesser, Lucas Mello Schnorr, Arnaud Legrand, Fabrice Dupros, and Philippe Olivier Alexandre Navaux. Using Simulation to Evaluate and Tune the Performance of Dynamic Load Balancing of an Over-Decomposed Geophysics Application. In *Proceedings of the European Conference on Parallel Processing*. Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors. Springer International Publishing, Cham, 2017, pages 192–205. ISBN: 978-3-319-64203-1.

[KBP95] Gerry Kingsley, Micah Beck, and James S. Plank. Compiler-assisted Checkpoint Optimization Using SUIF. In *Proceedings of the First SUIF Compiler Workshop*. Citeseer, 1995, pages 1–16.

[Kis02] Laszlo B. Kish. End of Moore's Law: Thermal (noise) Death of Integration in Micro and Nano Electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Toolset. In *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, 2008, pages 139–155.

[KRM+12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A Joint Performance Measurement Run-time Infrastructure for Periscope, Scalasca, Tau, and Vampir. In, *Tools for High Performance Computing*, pages 79–91. Springer, 2012.

[KLQ19] Douglas Kothe, Steven Lee, and Irene Qualters. Exascale Computing in the United States. *Computing in Science Engineering*, 21(1):17–29, 2019. ISSN: 1521-9615.

[KW85] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1985.

[KGP+18] Mohit Kumar, Saurabh Gupta, Tirthak Patel, Michael Wilder, Weisong Shi, Song Fu, Christian Engelmann, and Devesh Tiwari. Understanding and Analyzing Interconnect Errors and Network Congestion on a Large Scale HPC System. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018, pages 107–114.

[KGG+94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Volume 400. Benjamin/Cummings Redwood City, 1994.

[KNM+13] Chih-Song Kuo, Akihiro Nomura, Satoshi Matsuoka, Aamer Shah, Felix Wolf, and Ilya Zhukov. Environment Matters: How Competition for I/O Among Applications Degrades their Performance. *IPSJ SIG Technical Report*, (11):1–7, 2013.

[KSN+14] Chih-Song Kuo, Aamer Shah, Akihiro Nomura, Satoshi Matsuoka, and Felix Wolf. How File Access Patterns Influence Interference Among Cluster Applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2014, pages 185–193.

[LGZ+10] Zhiling Lan, Jiexing Gu, Ziming Zheng, Rajeev Thakur, and Susan Coghlan. A Study of Dynamic Meta-learning for Failure Prediction in Large-scale Systems. *Journal of Parallel and Distributed Computing*, 70(6):630–643, 2010.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 2004, page 75.

[LA18] Christopher T. Lee and Rommie E. Amaro. Exascale Computing: A New Dawn for Computational Biology. *Computing in Science Engineering*, 20(5):18–25, 2018. ISSN: 1521-9615.

[LF95] Chung-Chi Jim Li and W Kent Fuchs. Catch-compiler-assisted Techniques for Checkpointing. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing, 'Highlights from Twenty-Five Years'*. IEEE, 1995, page 213.

[LTS+gu] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C Sevcik. Locality and Loop Scheduling on NUMA Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, August 1993, pages 140–147.

[LN18] Matthias Lieber and Wolfgang E Nagel. Highly Scalable SFC-based Dynamic Load Balancing and its Application to Atmospheric Modeling. *Future Generation Computer Systems*, 82:575–590, 2018.

[LHG+08] John Christian Linford, Marc-André Hermanns, Markus Geimer, David Boehme, and Felix Wolf. Detecting Load Imbalance in Massively Parallel Applications. *Forschungszentrum Jülich, Tech. Rep. FZJ-JSC-IB-2008-09*, 2008.

[Luc92] Steven Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *ACM SIGPLAN Notices*. Volume 27. (7). ACM, 1992, pages 200–211.

[LV62] Robert E Lyons and Wouter Vanderkulk. The Use of Triple-modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[Man80] Benoit B. Mandelbrot. Fractal Aspects of the Iteration of $z \rightarrow \Lambda z$ (1-z) for Complex $\Lambda$ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[Max08] Don Maxwell. Restoring the CPA to CNL. *Proceedings of the CUG*, 2008.

[McC95] John D. McCalpin. Stream Benchmark. https://www.cs.virginia.edu/stream/. 1995.

[MBS+15] Rajat Mehrotra, Ioana Banicescu, Srishti Srivastava, and Sherif Abdelwahed. A Power-aware Autonomic Approach for Performance Management of Scientific Applications in a Data Center Environment. In, *Handbook on Data Centers*, pages 163–189, 2015.

[MH13] Yuangang Mei and Yuqing He. A New Spin-image Based 3D Map Registration Algorithm using Low Dimensional Feature Space. In *Proceedings of the 7th International Conference on Information and Automation*, 2013, pages 545–551.

[MNJ+15] Esteban Meneses, Xiang Ni, Terry Jones, and Don Maxwell. Analyzing the Interplay of Failures and Workload on a Leadership-class Supercomputer. *Computing*, 2(3):4, 2015.

[MJZ+12] Harshitha Menon, Nikhil Jain, Gengbin Zheng, and Laxmikant Kale. Automated Load Balancing Invocation Based on Application Characteristics. In *IEEE International Conference on Cluster Computing*, 2012, pages 373–381.

[MWZ+15] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. Adaptive Techniques for Clustered N-Body Cosmological Simulations. *Computational Astrophysics and Cosmology*, 2(1):1, 2015.

[MU49] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[MHH+05] Sarah E Michalak, Kevin W Harris, Nicolas W Hengartner, Bruce E Takala, and Stephen A Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, 2005.

[MB03] Amitabh Mishra and Prithviraj Banerjee. An Algorithm-based Error Detection Scheme for the Multigrid Method. *IEEE Transactions on Computers*, 52(9):1089–1099, 2003.

[MCC19] Ali Mohammed, Aurelien Cavelan, and Florina M. Ciorba. rDLB: A Novel Approach for Robust Dynamic Load Balancing of Scientific Applications with Independent Tasks. In *Proceedings of the International Conference on High Performance Computing and Simulation*, July 2019.

[MCC+19] Ali Mohammed, Aurélien Cavelan, Florina M. Ciorba, Rubén Cabezon, and Ioana Banicescu. Identifying Performance Challenges in Smoothed Particle Hydrodynamics Simulations. Poster at 2019 Platform for Advanced Scientific Computing Conference. July 2019.

[MCC+20] Ali Mohammed, Aurélien Cavelan, Florina M. Ciorba, Rubén Cabezon, and Ioana Banicescu. Two-level Dynamic Load Balancing for High Performance Scientific Applications. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*. Seattle, WA, USA, 2020.

[MC18a] Ali Mohammed and Florina M. Ciorba. A Study of the Performance of Scientific Applications with Dynamic Loop Scheduling under Perturbations. Poster at 2018 Platform for Advanced Scientific Computing Conference. July 2018.

[MC18b] Ali Mohammed and Florina M. Ciorba. SiL: An Approach for Adjusting Applications to Heterogeneous Systems Under Perturbations. In *Proceedings of the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms of the 24th International European Conference on Parallel and Distributed Computing*. Turin, August 2018.

[MC19a] Ali Mohammed and Florina M. Ciorba. Design of Robust Scheduling Methodologies in High Performance Computing. PhD Forum Poster at the 34th International Conference on High Performance Computing. July 2019.

[MC19b] Ali Mohammed and Florina M. Ciorba. SimAS: A Simulation-Assisted Approach for the Algorithm Selection Problem of Scheduling under Perturbations. *Concurrency and computation: practice and experience*:5648, 2019.

[MEC17a] Ali Mohammed, Ahmed Eleliemy, and Florina M. Ciorba. A Methodology for Bridging the Native and Simulated Execution of Parallel Applications. Poster at ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis. November 2017.

[MEC17b] Ali Mohammed, Ahmed Eleliemy, and Florina M. Ciorba. Towards the Reproduction of Selected Dynamic Loop Scheduling Experiments Using SimGrid-SimDag. Poster at IEEE International Conference on High Performance Computing and Communications. 2017.

[MEC18] Ali Mohammed, Ahmed Eleliemy, and Florina M. Ciorba. Performance Reproduction and Prediction of Selected Dynamic Loop Scheduling Experiments. In *Proceedings of the International Conference on High Performance Computing and Simulation*. Orléans, France, July 2018, page 8.

[MEC+18a] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. Experimental Verification and Analysis of Dynamic Loop Scheduling in Scientific Applications. In *Proceedings of the 17th International Symposium on Parallel and Distributed Computing*, 2018, page 8.

[MEC+18b] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. Experimental Verification and Analysis of Dynamic Loop Scheduling in Scientific Applications. *Arxiv e-prints*, April 2018.

[MEC+19] Ali Mohammed, Ahmed Eleliemy, Florina M. Ciorba, Franziska Kasielke, and Ioana Banicescu. A Methodology for Realistically Simulating the Performance of Scientific Applications on High Performance Computing Systems. *Future Generation Computer Systems Journal, On The Road to Exascale II Special Issue: Advances in High Performance Computing and Simulations.*, 2019.

[MKB+15] Mohammed Moness, Mahmoud Khaled, Mohammed Bakr, and Ali Mohammed. PID Control of a Lab Scale Single-Rotor Helicopter System using a Multicore Microcontroller. In *16th International Conference on Aerospace Sciences and Aviation Technology*. Cairo, Egypt, 2015.

[Moo+65] Gordon E Moore et al. Cramming More Components onto Integrated Circuits. 1965.

[Ni16] Xiang Ni. Mitigation of Failures in High Performance Computing via Runtime Techniques. PhD thesis. University of Illinois at Urbana-Champaign, 2016.

[PLS+17] Ayush Patwari, Ignacio Laguna, Martin Schulz, and Saurabh Bagchi. Understanding the Spatial Characteristics of DRAM Errors in HPC Clusters. In *Workshop on Fault-Tolerance for HPC at Extreme Scale*. ACM, 2017, pages 17–22.

[PPC86] Tang Peiyi and Yew Pen-Chung. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the International Conference on Parallel Processing*, 1986, pages 528–535.

[PIC+17] Pedro H Penna, Eduardo C Inacio, Márcio Castro, Patrícia Plentz, Henrique C Freitas, François Broquedis, and Jean-François Méhaut. Assessing the Performance of the SRR Loop Scheduler with Irregular Workloads. *Procedia Computer Science*, 108:255–264, 2017.

[PCP+17] Pedro Henrique Penna, Márcio Castro, Patrícia Plentz, Henrique Cota de Freitas, François Broquedis, and Jean-François Méhaut. BinLPT: A Novel Workload-aware Loop Scheduler for Irregular Parallel Loops. In *Anais do XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*. SBC, 2017.

[PBG+85] Gregory F. Pfister, William Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfelder, Kevin P. McAuliffe, Evelin Melton, Alan Norton, and Jodi Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the International Conference on Parallel Processing*, 1985, pages 764–772.

[Phe08] Chuck Pheatt. Intel® Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[PBW+05] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kale, and Klaus Schulten. Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[PLF+07] Mark M Phillips, Weidong Li, Joshua A Frieman, SI Blinnikov, Darren DePoy, José L Prieto, P Milne, Carlos Contreras, Gastón Folatelli, Nidia Morrell, Mario Hamuy, Nicholas B. Suntzeff, Miguel Roth, Sergio González, Wojtek Krzeminski, Alexei V. Filippenko, Wendy L. Freedman, Ryan Chornock, Saurabh Jha, Barry F. Madore, S. E. Persson, Christopher R. Burns, Pamela Wyatt, David Murphy, Ryan J. Foley, Mohan Ganeshalingam, Franklin J. D. Serduke, Kevin Krisciunas, Bruce Bassett, Andrew Becker, Ben Dilday, Peter M. Eastman J. Garnavich, Jon Holtzman, Richard Kessler, Hubert Lampeitl, John Marriner, S. Frank, J. L. Marshall, Gajus Miknaitis, Masao Sako26, Donald P. Schneider, Kurt van der Heyden, and Naoki Yasuda. The Peculiar SN 2005hk: Do Some type Ia Supernovae Explode as Deflagrations? *Publications of the Astronomical Society of the Pacific*, 119(854):360, 2007.

[PBK95] James S Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted Memory Exclusion for Fast Checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[PLP98] James S Plank, Kai Li, and Michael A Puening. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[PK87] Constantine D Polychronopoulos and David J Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 100(12):1425–1439, 1987.

[PNW18] Suraj Prabhakaran, Marcel Neumann, and Felix Wolf. Efficient Fault Tolerance Through Dynamic Node Replacement. In *Proceedings of the 18th*

*IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2018, pages 163–172.

[QL18] Depei Qian and Zhongzhi Luan. High Performance Computing Development in China: A Brief Review and Perspectives. *Computing in Science & Engineering*, 21(1):6–16, 2018.

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pages 427–436.

[Raw00] James B. Rawlings. Tutorial: Overview of Model Predictive Control. *IEEE Control Systems*, 20(3):38–52, 2000.

[Ric76] John R Rice. The Algorithm Selection Problem. In, *Advances in Computers*. Volume 15, pages 65–118. Elsevier, 1976.

[RBB+12] Arun F Rodrigues, Keren Bergman, David P Bunde, Elliot Cooper-Balis, Kurt Brian Ferreira, Karl Scott Hemmert, Brian Barrett, Cassandra Versaggi, Robert Hendry, Bruce Jacob, Hyesoon Kim, Vitus Joseph Leung, Michael J. Levenhagen, Mitchelle Rasquinha, Rolf Riesen, Paul Rosenfeld, Maria del Carmen Ruiz Varela, and Sudhakar Yalamanchili. Improvements to the Structural Simulation Toolkit. Technical report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2012.

[Rus16] John Russell. How to Kill a Supercomputer, Tips from an Expert. https://www.hpcwire.com/2016/02/24/how-to-kill-a-supercomputer-tips-from-an-expert. [Online; accessed 14 March 2018]. 2016.

[Sch97] Robert R. Schaller. Moore's Law: Past, Present and Future. *IEEE Spectrum*, 34(6):52–59, 1997. ISSN: 0018-9235.

[SG10] Bianca Schroeder and Garth Gibson. A Large-scale Study of Failures in High-performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010.

[SG07] Bianca Schroeder and Garth A Gibson. Understanding Failures in Petascale Computers. In *Journal of Physics: Conference Series*. Volume 78. (1). IOP Publishing, 2007, page 012022.

[SPW11] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: a Large-scale Field Study. *Communications of the ACM*, 54(2):100–107, 2011.

[SBW+19] Thomas C Schulthess, Peter Bauer, Nils Wedi, Oliver Fuhrer, Torsten Hoefler, and Christoph Schär. Reflecting on the Goal and Baseline for Exascale Computing: A Roadmap Based on Weather and Climate Simulations. *Computing in Science Engineering*, 21(1):30–41, 2019. ISSN: 1521-9615.

[Ser19] ServeTheHome. ORNL Summit Supercomputer Architecture. https://www.servethehome.com/top500-june-2018-edition-new-systems-analysis/ornl-summit-supercomputer-architecture. [Online; accessed 25 September 2019]. September 2019.

[SKN+19] Aamer Shah, Chih-Song Kuo, Akihiro Nomura, Satoshi Matsuoka, and Felix Wolf. How File-access Patterns Influence the Degree of I/O Interference between Cluster Applications. *Supercomputing Frontiers and Innovations*, 6(2):29–55, 2019.

[SMW18] Aamer Shah, Matthias Müller, and Felix Wolf. Estimating the Impact of External Interference on Application Performance. In *Proceedings of the European Conference on Parallel Processing*. Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors. Springer International Publishing, 2018, pages 46–58. ISBN: 978-3-319-96983-1.

[SWZ+13] Aamer Shah, Felix Wolf, Sergey Zhumatiy, and Vladimir Voevodin. Capturing Inter-application Interference on Clusters. In *Proceedings of the International Conference on Cluster Computing*. IEEE, 2013, pages 1–5.

[SSR12] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2012, pages 69–78.

[Sim14] SimGrid. SimGrid Calibration documentation. http://simgrid.gforge.inria.fr/contrib/smpi-calibration-doc/. [Online; accessed 17 April 2018]. 2014.

[SK05] David Skinner and William Kramer. Understanding the Causes of Performance Variability in HPC Workloads. In *The International IEEE Workload Characterization Symposium*, 2005, pages 137–149.

[SB01] Lorna Smith and Mark Bull. Development of Mixed Mode MPI/OpenMP Applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[SWA+14] Marc Snir, Robert W. Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan A. Debardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing Failures in Exascale Computing. *International Journal High Performance Computing Applications*, 28(2):129–173, 2014.

[Sor19] Bob Sorensen. Japan's Flagship 2020 "Post-K" System. *Computing in Science & Engineering*, 21(1):48–49, 2019.

[SDB+15] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, the Bad, and the Ugly. *ACM SIGPLAN Notices*, 50(4):297–310, 2015.

[SBC10] Srishti Srivastava, Ioana Banicescu, and Florina M. Ciorba. Investigating the Robustness of Adaptive Dynamic Loop Scheduling on Heterogeneous Computing Systems. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium Workshops and PhD Forum , 11th International Workshop on Parallel and Distributed Scientific and Engineering Computing*, April 2010, pages 1–8.

[SCB10] Srishti Srivastava, Florina M. Ciorba, and Ioana Banicescu. Employing a Study of the Robustness Metrics to Assess the Reliability of Dynamic Loop Scheduling. In *Proceedings of the Student Research Symposium of the 17th IEEE High Performance Computing Conference*, December 2010, 5p.

[SSB+12] Srishti Srivastava, Nitin Sukhija, Ioana Banicescu, and Florina M. Ciorba. Analyzing the Robustness of Dynamic Loop Scheduling for Heterogeneous Computing Systems. In *Proceedings of the 11th International Symposium on Parallel and Distributed Computing*. IEEE, 2012, pages 156–163.

[STL+15] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-based Runtime System for Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015.

[SAB+19] Sauro Succi, Giorgio Amati, M. Bernaschi, Giacomo Falcucci, Marco Lauricella, and Andrea Montessori. Towards Exascale Lattice Boltzmann Computing. *Computers & Fluids*, 2019.

[SBC15] Nitin Sukhija, Ioana Banicescu, and Florina M. Ciorba. Investigating the Resilience of Dynamic Loop Scheduling in Heterogeneous Computing Systems. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, 2015, pages 194–203.

[SBS+13a] Nitin Sukhija, Ioana Banicescu, Srishti Srivastava, and Florina M. Ciorba. Evaluating the Flexibility of Dynamic Loop Scheduling on Heterogeneous Systems in the Presence of Fluctuating Load Using SimGrid. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, 2013, pages 1429–1438.

[SBS+13b] Nitin Sukhija, Ioana Banicescu, Srishti Srivastava, and Florina M. Ciorba. Evaluating the Flexibility of Dynamic Loop Scheduling on Heterogeneous Systems in the Presence of Fluctuating Load Using SimGrid. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium Workshops*, 2013, pages 1429–1438. ISBN: 9780769549798.

[SMS+14] Nitin Sukhija, Brandon Malone, Srishti Srivastava, Ioana Banicescu, and Florina M. Ciorba. Portfolio-based Selection of Robust Dynamic Loop Scheduling Algorithms Using Machine Learning. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium Workshops*. Phoenix, AZ, USA, 2014, pages 1638–1647.

[TMT+19] Giuliano Taffoni, Giuseppe Murante, Luca Tornatore, David Goz, Stefano Borgani, Manolis Katevenis, Nikolaos Chrysos, and Manolis Marazakis. Shall Numerical Astrophysics Step into the Era of Exascale Computing? *Arxiv preprint arxiv:1904.11720*, 2019.

[TSK+16] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. New-sum: A Novel Online ABFT Scheme for General Iterative Methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pages 43–55.

[TSL+17] Rafael Keller Tesser, Lucas Mello Schnorr, Arnaud Legrand, Fabrice Dupros, and Philippe Olivier Alexandre Navaux. Using Simulation to Evaluate and Tune the Performance of Dynamic Load Balancing of

an Over-decomposed Geophysics Application. In *Proceedings of the International Conference on Parallel and Distributing Computing*, 2017, pages 192–205.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[TN93] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. In SOCC '13, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1.

[VL09] Pedro Velho and Arnaud Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009, page 10.

[VLWB+19] Verónica G. Vergara Larrea, Joubert Wayne, Michael J. Brim, Reuben D. Budiardja, Don Maxwell, Matt Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, Chris Fuson, Daniel Pelfrey, Oscar Hernandez, Dustin Leverman, Jesse Hanley, Mark Berrill, and Arnold Tharrington. Scaling the Summit: Deploying the World Fastest Supercomputer. In *International Workshop on OpenPOWER for HPC*. Resolution Publication, 2019, page 20.

[WYC+11] Rui Wang, Erlin Yao, Mingyu Chen, Guangming Tan, Pavan Balaji, and Darius Buntinas. Building Algorithmically Nonstop Fault Tolerant MPI Programs. In *Proceedings of the 18th IEEE International Conference on High Performance Computing*, 2011, pages 1–9.

[WJS+13] Yizhuo Wang, Weixing Ji, Feng Shi, and Qi Zuo. A Work-stealing Scheduling Framework Supporting Fault Tolerance. In *Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pages 695–700.

[WNC+12] Yizhuo Wang, Alexandru Nicolau, Rosario Cammarota, and Alexander V. Veidenbaum. A Fault Tolerant Self-scheduling Scheme for Parallel Loops on Shared Memory Systems. In *Proceedings of the IEEE 19th International Conference on High Performance Computing*, 2012, pages 1–10.

[WLB+16] Carsten Weinhold, Adam Lackorzynski, Jan Bierbaum, Martin Küttler, Maksym Planeta, Hermann Härtig, Amnon Shiloh, Ely Levy, Tal Ben-Nun, Amnon Barak, Thomas Steinke, Thorsten Schütt, Jan Fajerski, Alexander Reinefeld, Matthias Lieber, and Wolfgang E. Nagel. FFMK: A Fast and Fault-tolerant Microkernel-based System for Exascale Computing. In, *Software for Exascale Computing-SPPEXA 2013-2015*, pages 405–426. Springer, 2016.

[WKK+18] Jeremiah J. Wilke, Joseph P. Kenny, Samuel Knight, and Sebastien Rumley. Compiler-Assisted Source-to-Source Skeletonization of Application Models for System Simulation. In *High Performance Computing*. Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors. Springer International Publishing, Cham, 2018, pages 123–143. ISBN: 978-3-319-92040-5.

[WYL+12] Chao-Chin Wu, Chao-Tung Yang, Kuan-Chou Lai, and Po-Hsun Chiu. Designing Parallel Loop Self-scheduling Schemes Using the Hybrid MPI and OpenMP Programming Model for Multi-core Grid Systems. *The Journal of Supercomputing*, 59(1):42–60, 2012.

[YC03] Yang Yang and Henri Casanova. RUMR: Robust Scheduling for Divisible Workloads. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003, pages 114–123.

[Yes16] Aram Yesildeniz. Visualisation of Scheduling Algorithms. https://hpc.dmi.unibas.ch/HPC/Completed_ Theses_ and_ Projects_ files/project-report.pdf. [Online; accessed 24 July 2019]. 2016.

[Yil19] Akan Yilmaz. Implementation of Scheduling Algorithms in an OpenMP RuntimeLibrary. Master Thesis. 2019.

[YMM13a] Hassan Youness, Ali Mohammed, and Mohammed Moness. Fault Tolerant Heterogeneous MPSOC Schedule Length Minimization Based on Platform Reliability. In *Japan-Egypt International Conference on Electronics, Communications and Computers*. Cairo, Egypt, 2013.

[YMM13b] Hassan Youness, Ali Mohammed, and Mohammed Moness. Fault Tolerant Heterogeneous Scheduling for Precedence Constrained Task Graphs using Simulated Annealing. In *8th International Conference on Computer Engineering and Systems*. Cairo, Egypt, 2013.

[You74] John W Young. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM*, 17(9):530–531, 1974.